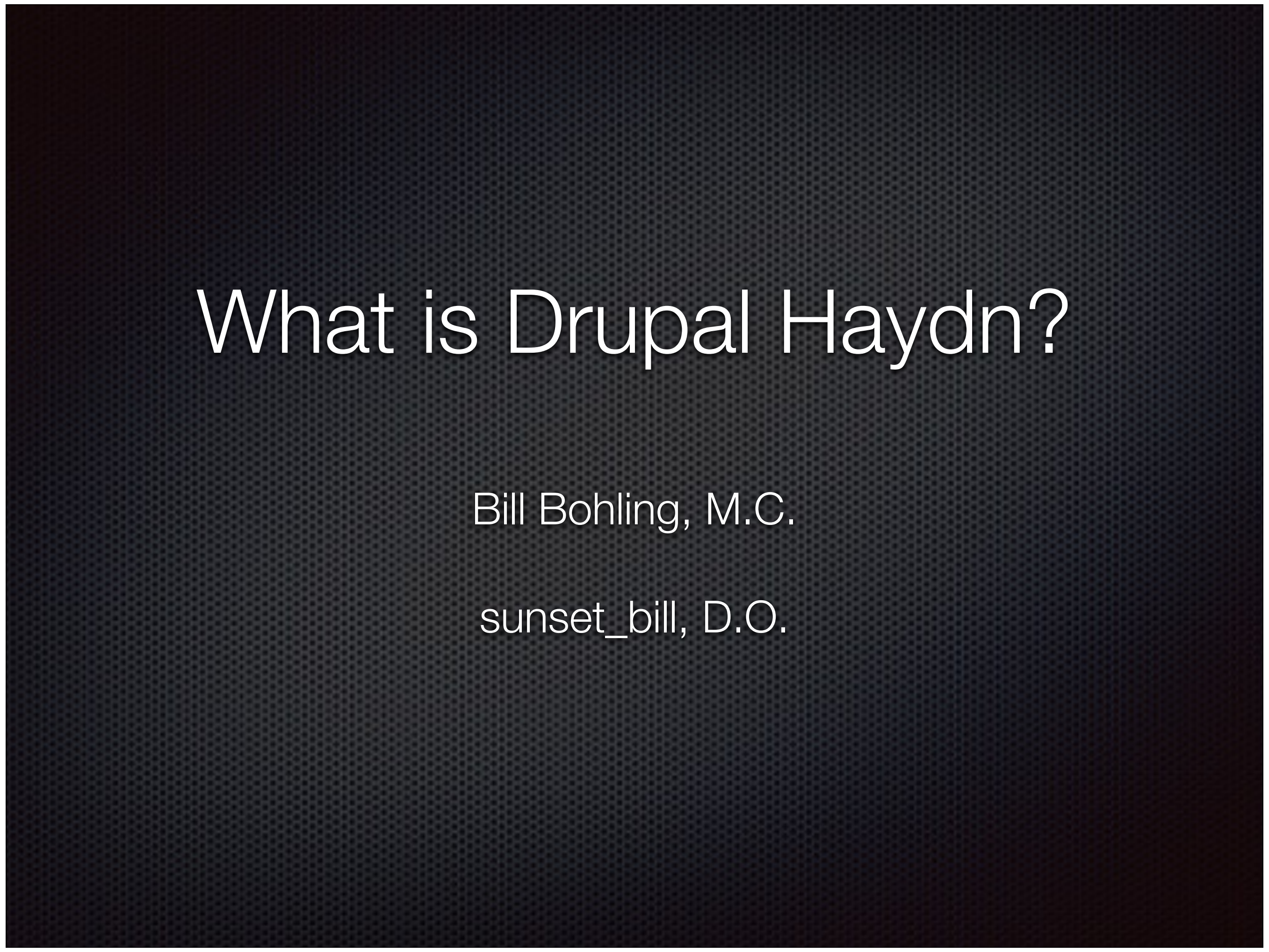


What is Drupal Haydn?

Bill Bohling, M.C.

sunset_bill, D.O.

Welcome to What is Drupal Haydn.

I'm your host, Bill Bohling. My Drupal id is sunset_bill.

I am not a Symfony developer, I do Drupal at Turner. I first heard about Symfony and Drupal when Dries announced it at Drupalcon Denver, but I only got really curious about how they worked together after a session at this year's Drupalcon given by Ryan Weaver, who is the Docs lead for Symfony.

The group I work with gets together weekly for our own version of TED talks, and I decided that would be a good way to learn some Symfony stuff. So I'd come in on talk day, pick a Symfony component to look at, dig into the documentation, and put together whatever kind of talk I could come up with in 3 hours or less. I called them Surprise Symfony because I'd have no idea what I was going to talk about, and I didn't tell anybody when I did know.

People seemed to like them, so I've polished them up a bit and put them together into what you're about to hear today.

I will warn you that this is going to be a bit dense. Pretty much every component I'm going to talk about is worthy of its own presentation, but all you get are some code snippets. My goal here today is to just to pull back the curtain a little and give you an idea of what's there. Don't panic. I'll be making these slides available on the camp site, and we'll always have the video.

So, what is Drupal Haydn? It's a pun.

Surprise Symphony, H. 1/94

Josef Haydn, Rohrau, Austria, 1732-1809

Allow me to introduce Josef Haydn.

He mentored Mozart and was one of Beethoven’s teachers. Perhaps that’s why he’s called “Father of the Symphony”.

His Symphony #94 is called the Surprise Symphony, but I’ll let you google that on your own.

So much for classical music.

Symfony

Fabien Potencier, Paris, France, present day

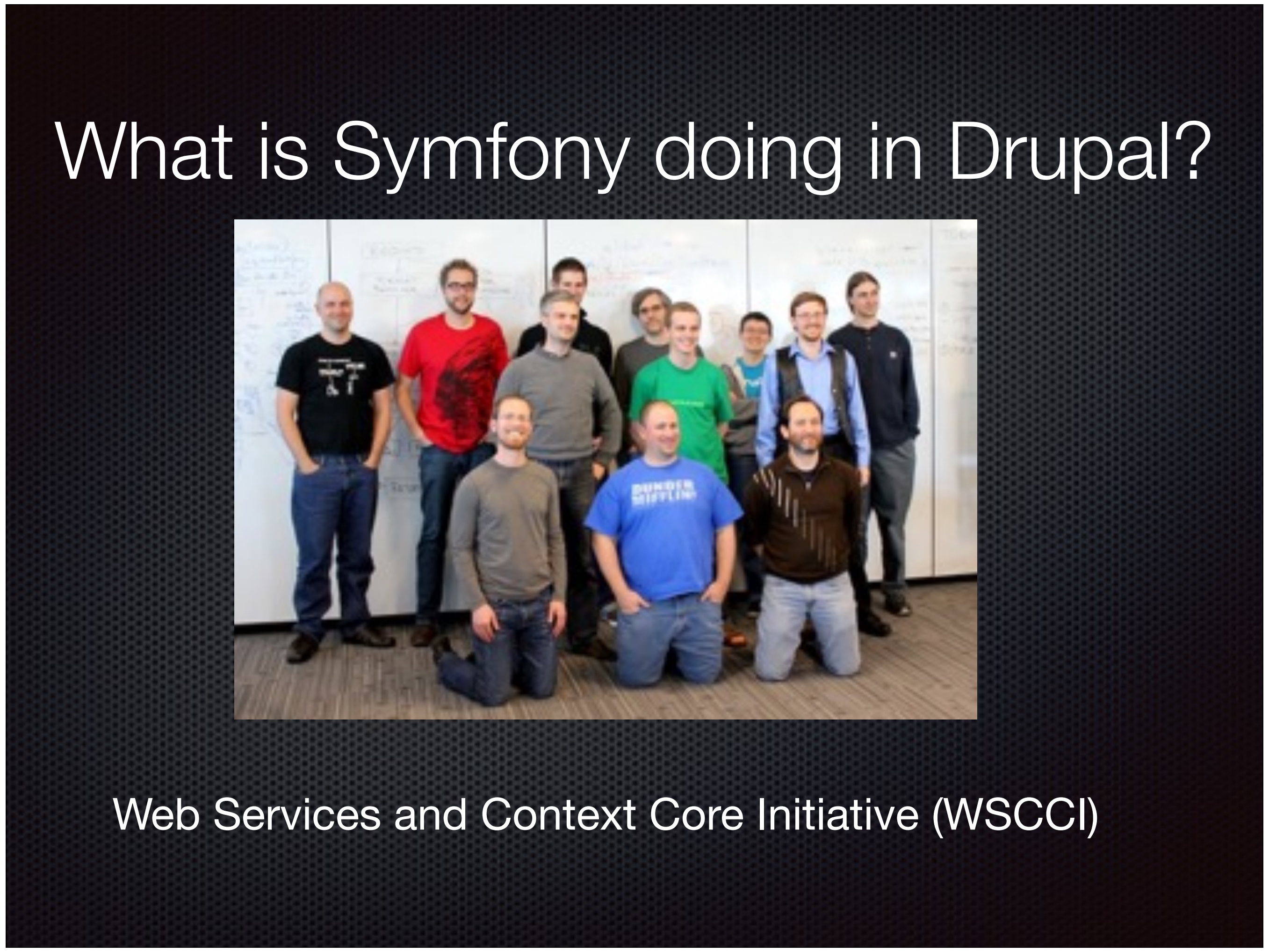
This is Fabien Potencier, the father of Symfony.

He started the Symfony project in 2004 as a way to develop better websites faster. In October, 2005, Symfony was released as free software under the MIT license.

So, what is Symfony, anyway? The website says that "Symfony is a set of PHP Components, a Web Application framework, a Philosophy, and a Community — all working together in harmony."

Or, as Ryan Weaver put it, Symfony is the reason we had to wait 3 years for Drupal 8.

And who do we have to thank for that?



What is Symfony doing in Drupal?

Web Services and Context Core Initiative (WSCCI)

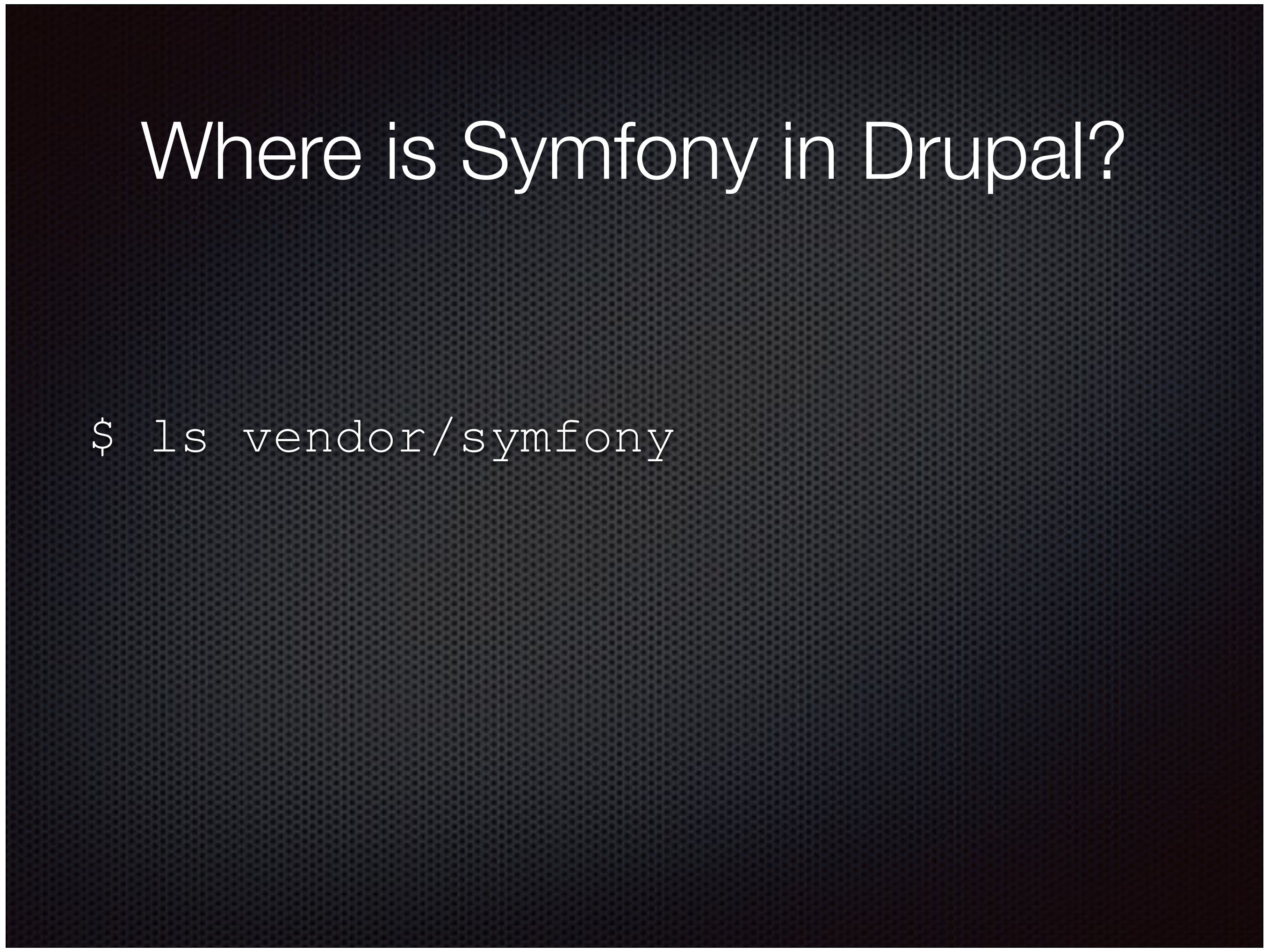
These folks.

In Feb. 2012 they all got together to discuss the Web Services and Context Core Initiative (WSCCI) for Drupal 8. They realized this was going to involve 3 major areas: Web Services, layouts and turning the underlying toolset into a more loosely-coupled framework. Rather than try to address all of them, they decided that a good RESTful architecture would make the other things possible, so the initiative was scaled back to just address web services.

One of the attendees was Fabien Potencier, and he pointed out that several Symfony 2 components were already doing what Drupal needed, so rather than trying to emulate that, the decision was made to incorporate Symfony 2 into Drupal 8.

Last I looked, Symfony provides 40-some discrete components. Drupal uses around 20 of these. I'm not going to try to cover all of them, the goal here is just to make you aware that they're there so you can take advantage of them to make your life easier.

I'll be going into some detail about a couple of core components and few components that my group has found to be useful.



Where is Symfony in Drupal?

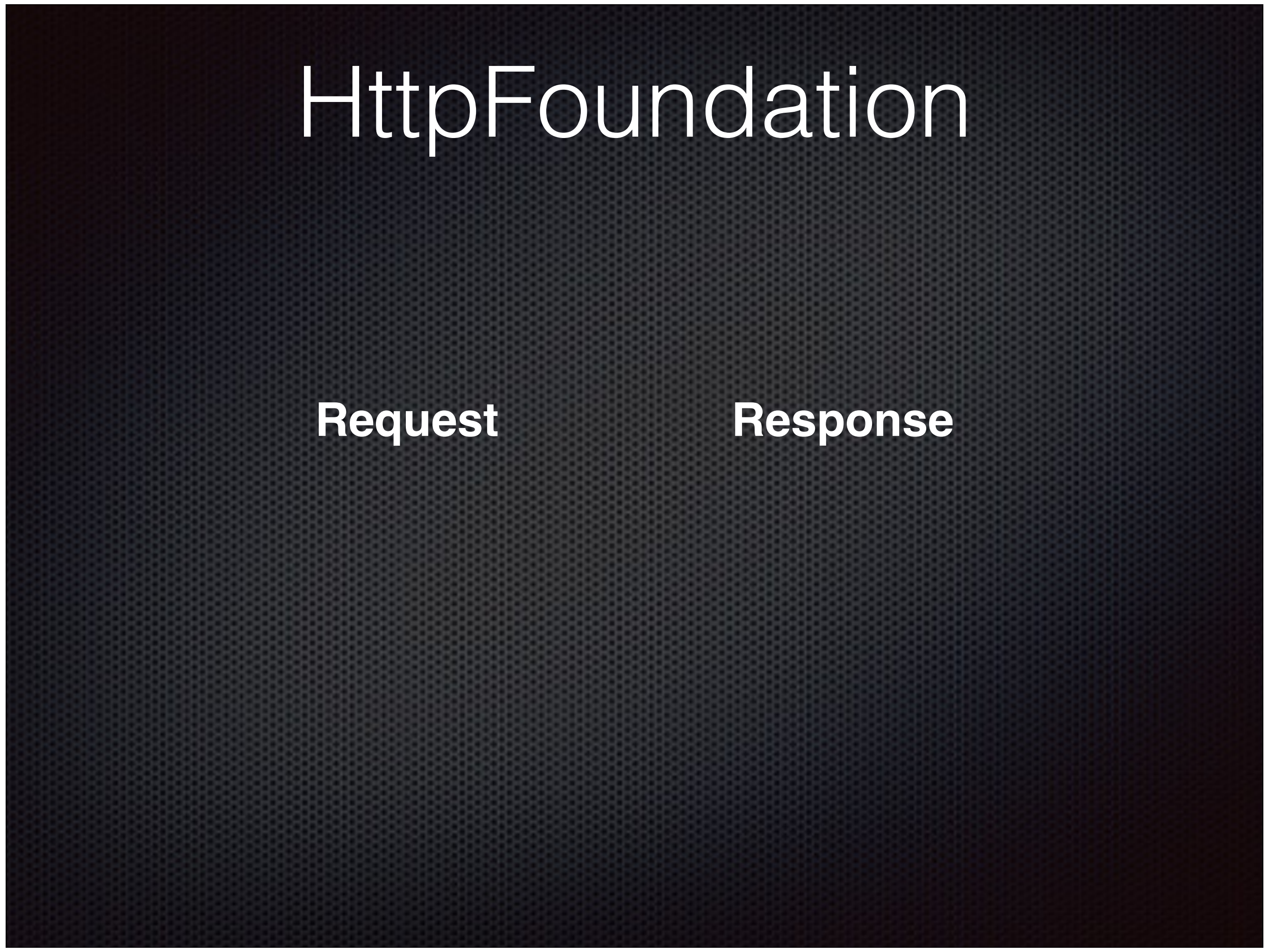
```
$ ls vendor/symfony
```

Where is Symfony?

Everything I'll be talking about today ships with Drupal and can be found in your installation's vendor/symfony directory after you've run composer install.

They're just sitting there. Waiting for you to use them.

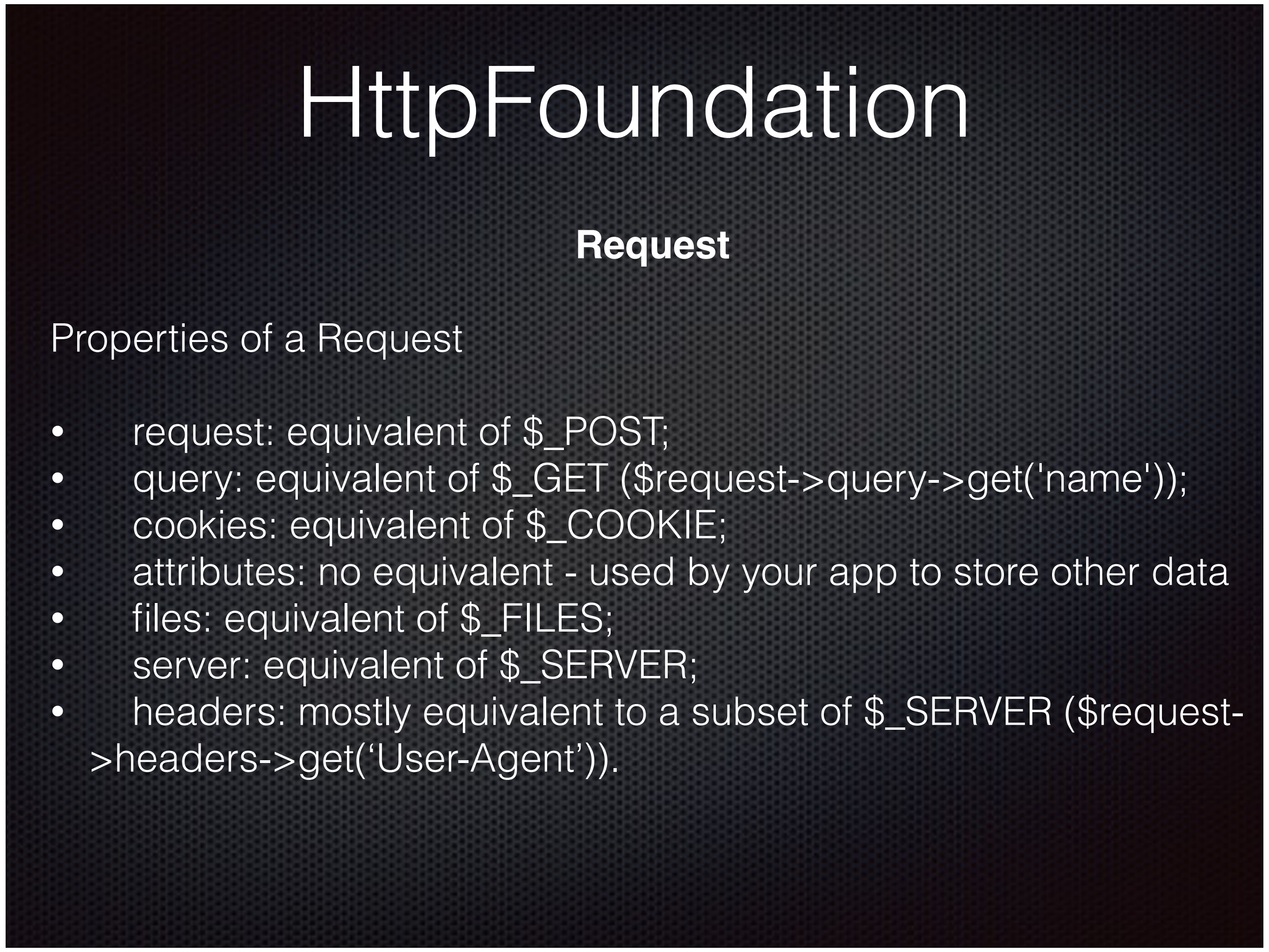
So, on to the fun stuff.



HttpFoundation

Request **Response**

The first component we're going to look at is HttpFoundation. This is one is a major reason why Symfony is in D8. What it basically does is provide Request and Response classes and lots of methods for dealing with them. It does such a good job of this that Drupal uses these as-is.



HttpFoundation

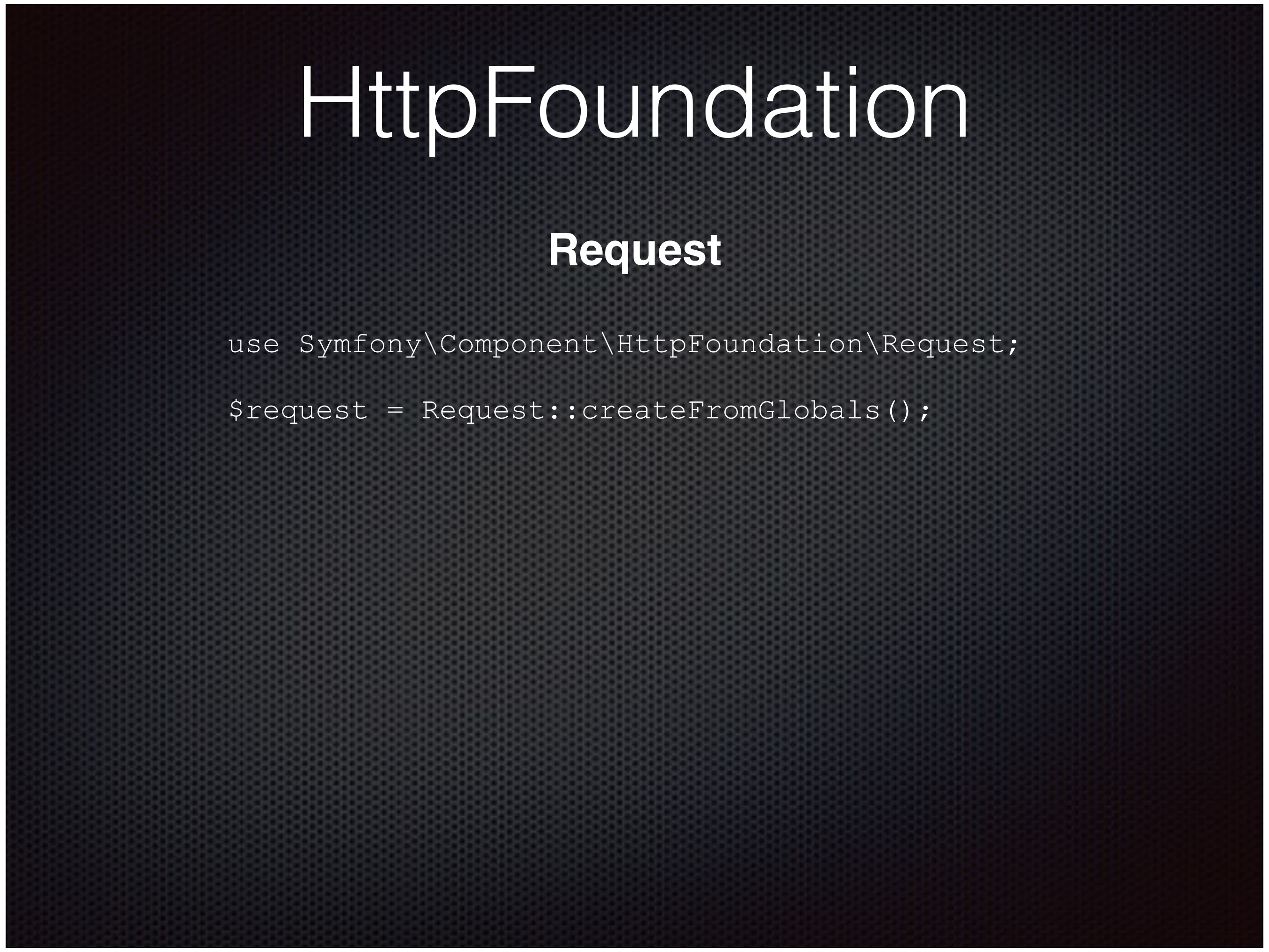
Request

Properties of a Request

- request: equivalent of $_POST;
- query: equivalent of $_GET ($request->query->get('name'));
- cookies: equivalent of $_COOKIE;
- attributes: no equivalent - used by your app to store other data
- files: equivalent of $_FILES;
- server: equivalent of $_SERVER;
- headers: mostly equivalent to a subset of $_SERVER ($request->headers->get('User-Agent')).

So, let's start with the Request.

The properties of a request are the PHP request globals, plus an attributes array that your app can use to pass along other data. HttpFoundation turns all of these into objects for us, so they're really easy to work with.

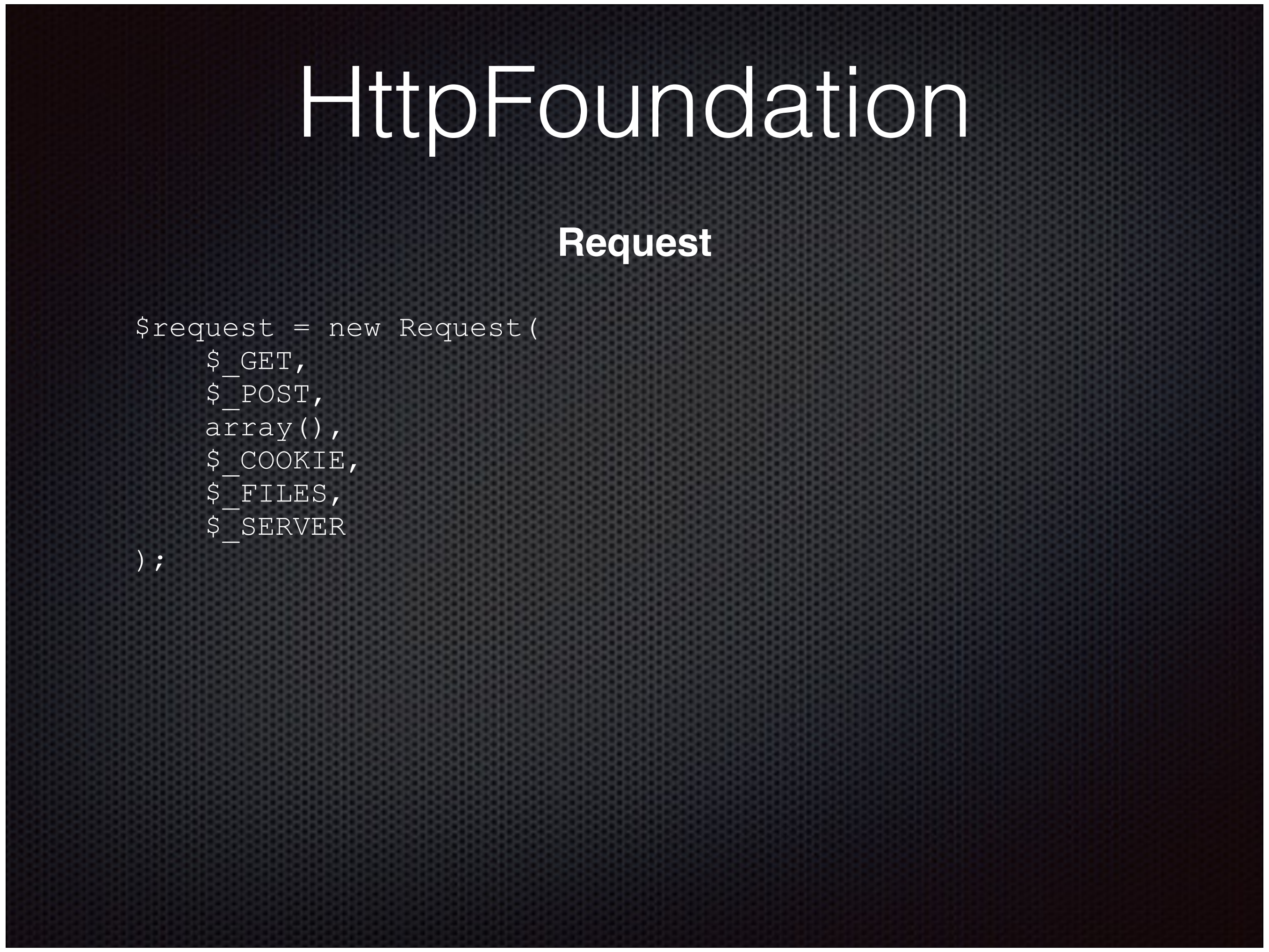


HttpFoundation

Request

```
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();
```

To get a Request object, you can use Symfony's createFromGlobals() method. In fact, this is exactly what Drupal does, as we'll see in a bit.



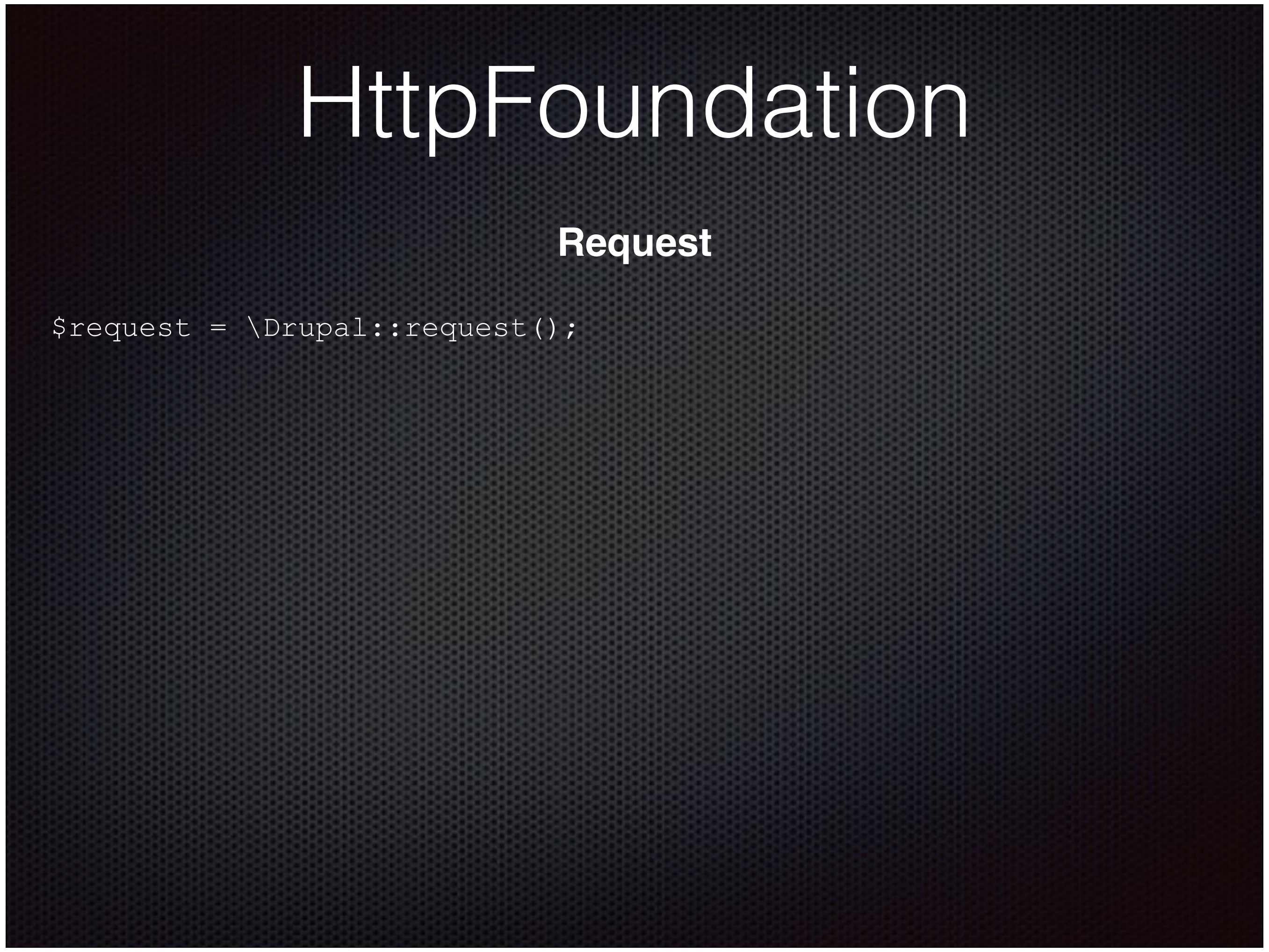
HttpFoundation

Request

```
$request = new Request(
    $_GET,
    $_POST,
    array(),
    $_COOKIE,
    $_FILES,
    $_SERVER
);
```

If you like typing, you can specify the request properties yourself in Request constructor arguments.

Here’s the long-hand version of createFromGlobals().



HttpFoundation

Request

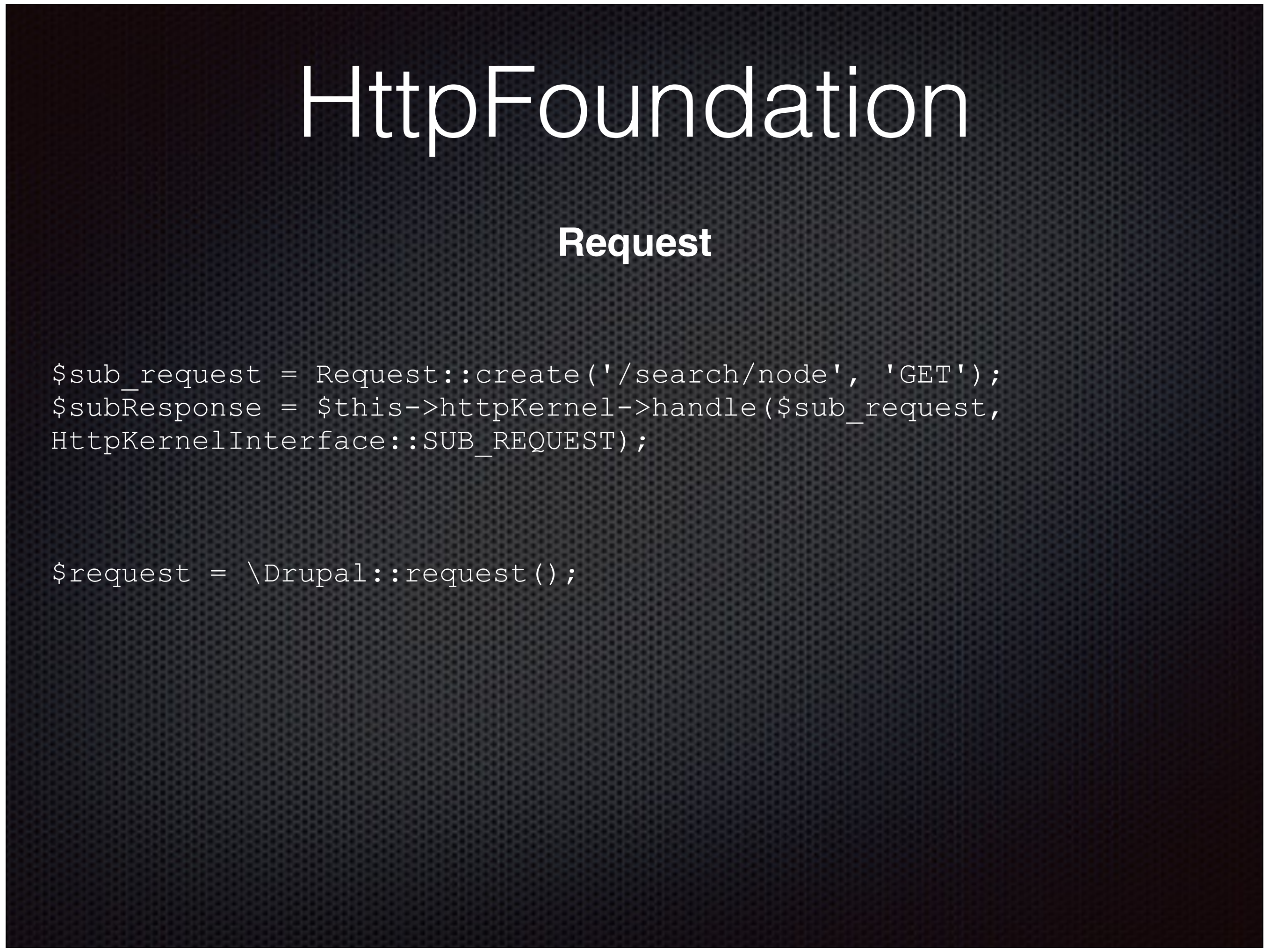
```
$request = \Drupal::request();
```

But we don’t have to do either of those.

Since Drupal has already called createFromGlobals(), you can get a request object by calling Drupal’s request() method.

You can do that. But even the Drupal::request() documentation says you shouldn’t. Any ideas why not?

A big reason is that this only works for HTTP requests. It won’t work in unit tests, where Drupal isn’t running. It also won’t work for all queue processors. And it’s not reliable for command-line invocations, so I don’t know about drush commands.



HttpFoundation

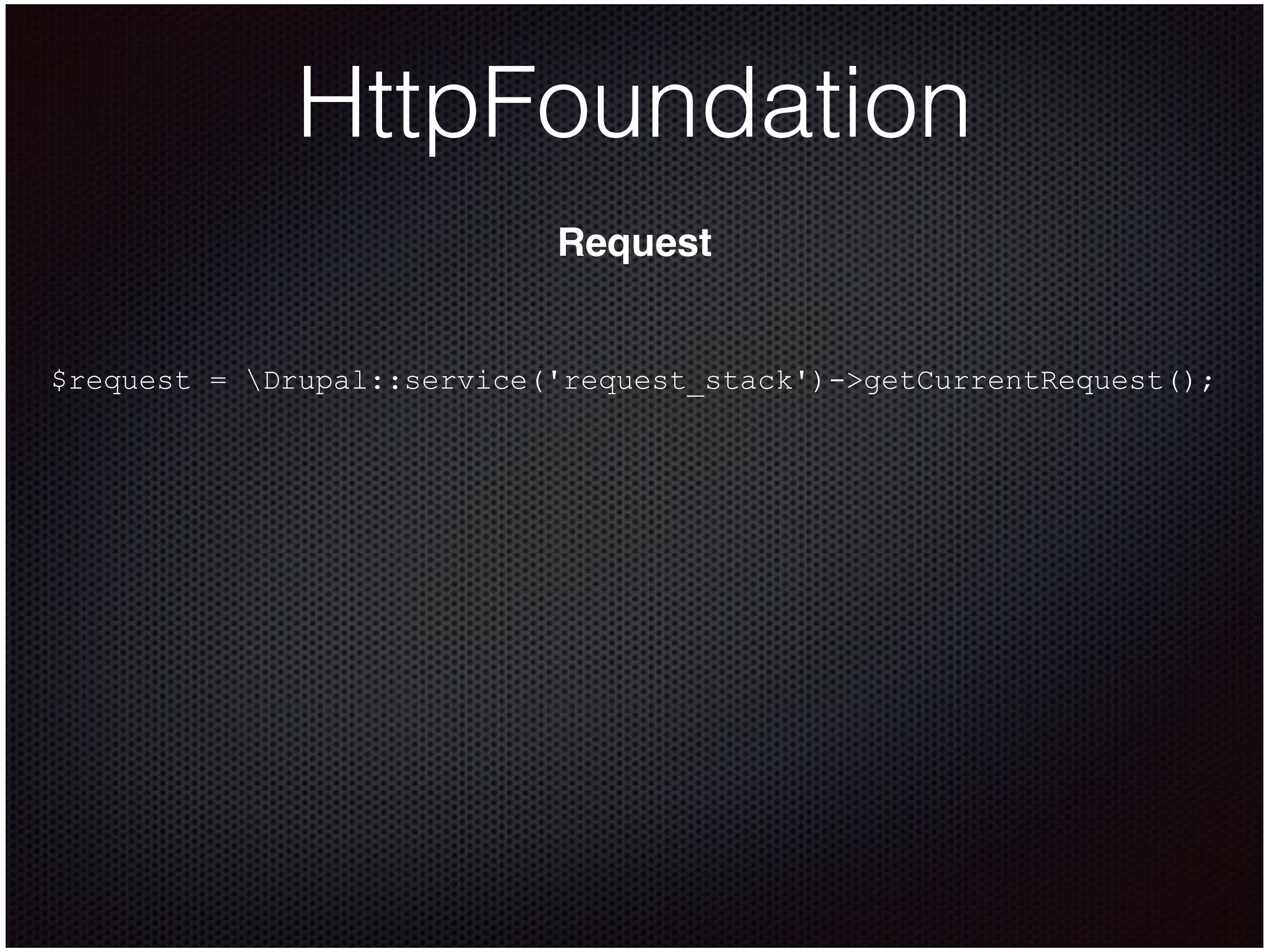
Request

```
$sub_request = Request::create('/search/node', 'GET');
$subResponse = $this->httpKernel->handle($sub_request,
HttpKernelInterface::SUB_REQUEST);



$request = \Drupal::request();
```

Most importantly, the request instance can change during the container's lifetime.

What if someone has generated a subrequest in the current container, like we see here?

If you call Drupal::request() in a case like this, you can't be sure which instance of Request you're getting. It might be the master request or a parent request. If you're lucky, it might even be the current request.



HttpFoundation

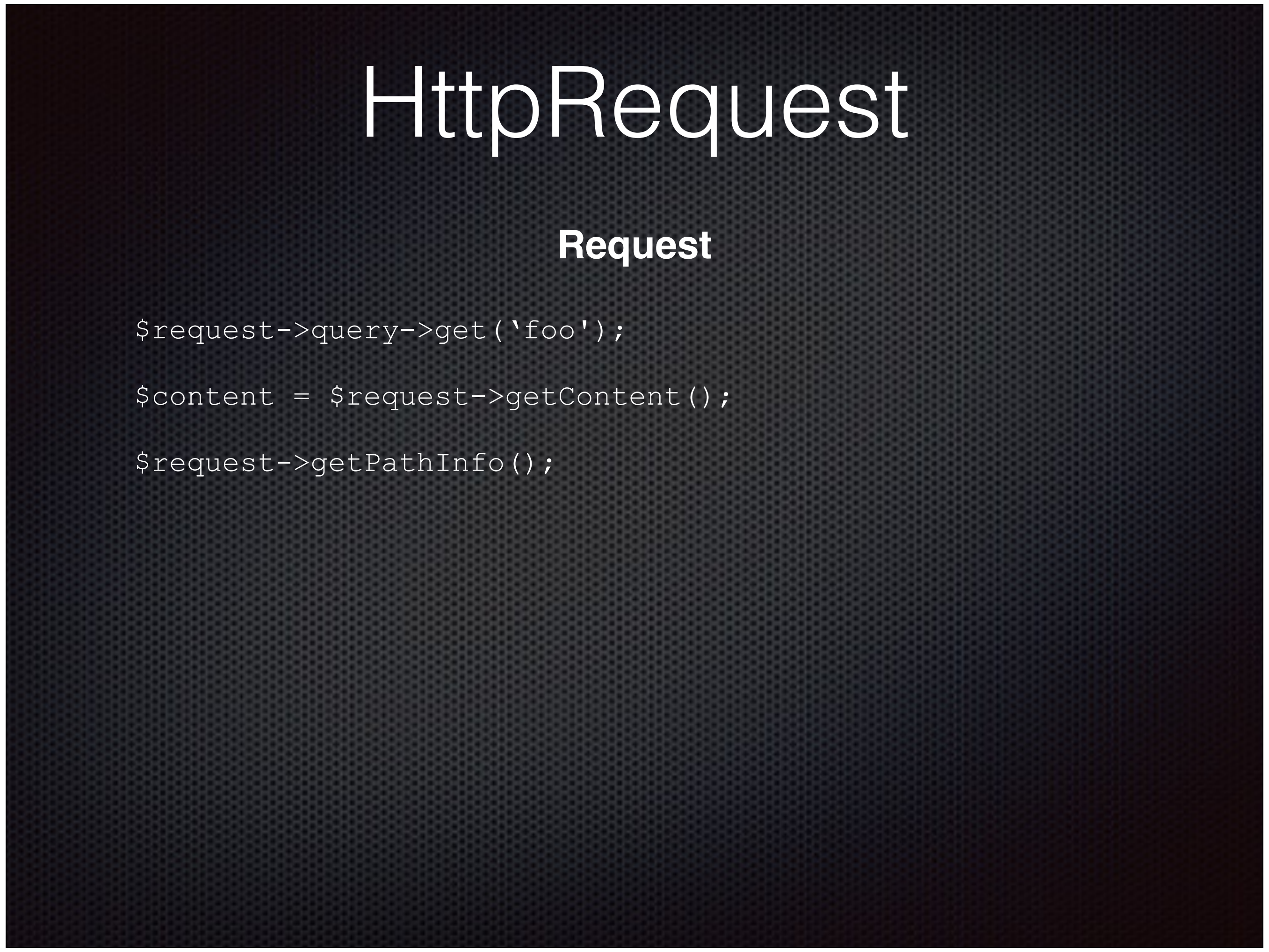
Request

```
$request = \Drupal::service('request_stack')->getCurrentRequest();
```

The Drupal::request docs say it’s better to get the request from a service, and what do you know, Drupal provides a request_stack service. So here’s the most reliable way to get the current request.

The request_stack service wraps the Symfony RequestStack component, which is a class in HttpFoundation designed to handle this very problem.

RequestStack controls the lifecycle of a request. This means that as long as you call getCurrentRequest from the request stack, you will get the current request. RequestStack also provides getMasterRequest() and getParentRequest() methods if you really need something other than the current request.



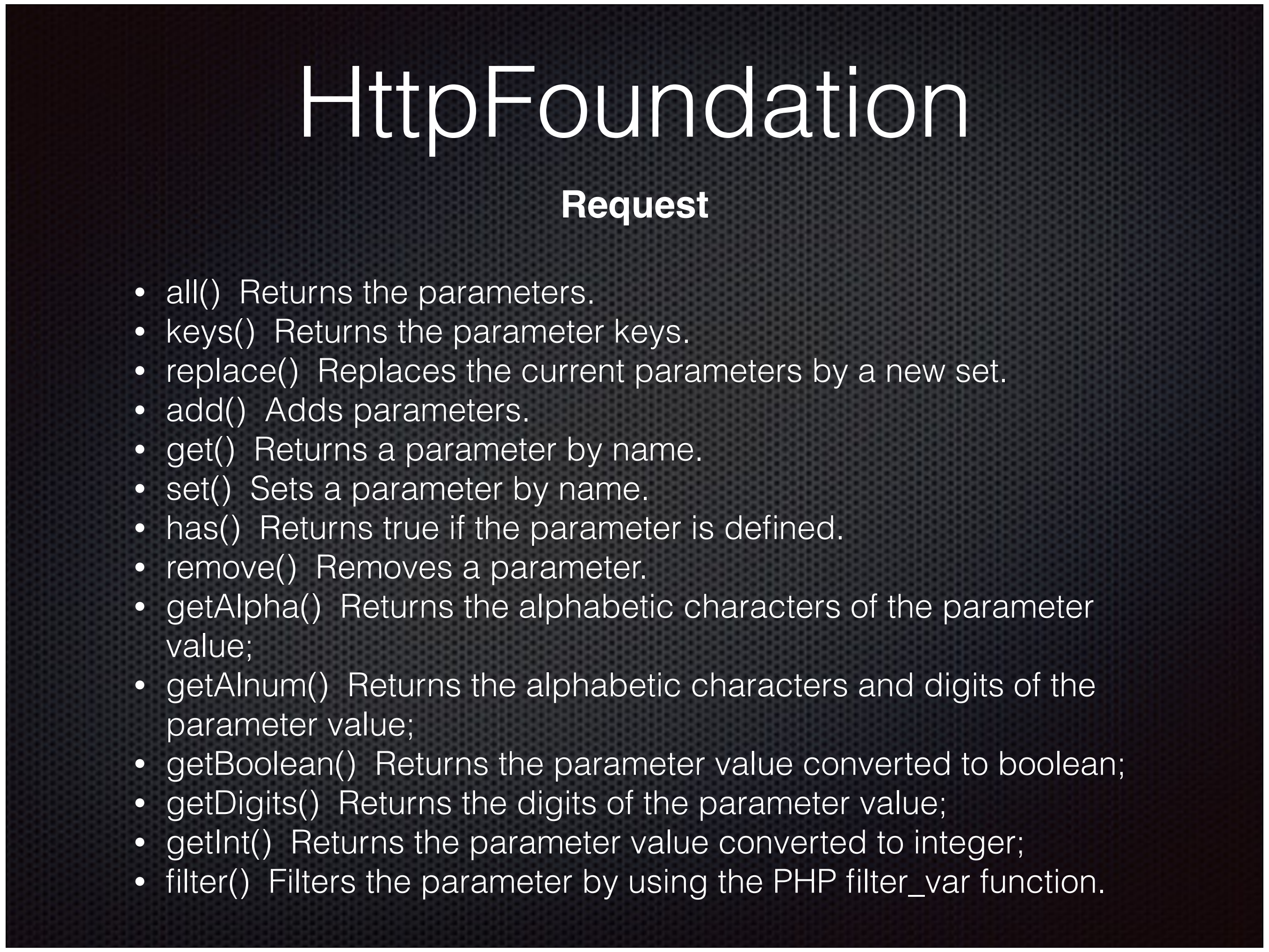
HttpRequest

Request

```
$request->query->get('foo');

$content = $request->getContent();

$request->getPathInfo();
```

Once you’ve got your request object, you can start calling methods on it.

The Symfony Request class provides far too many methods to cover here, so I’ll refer you to the Request class docs on D.O. for details.

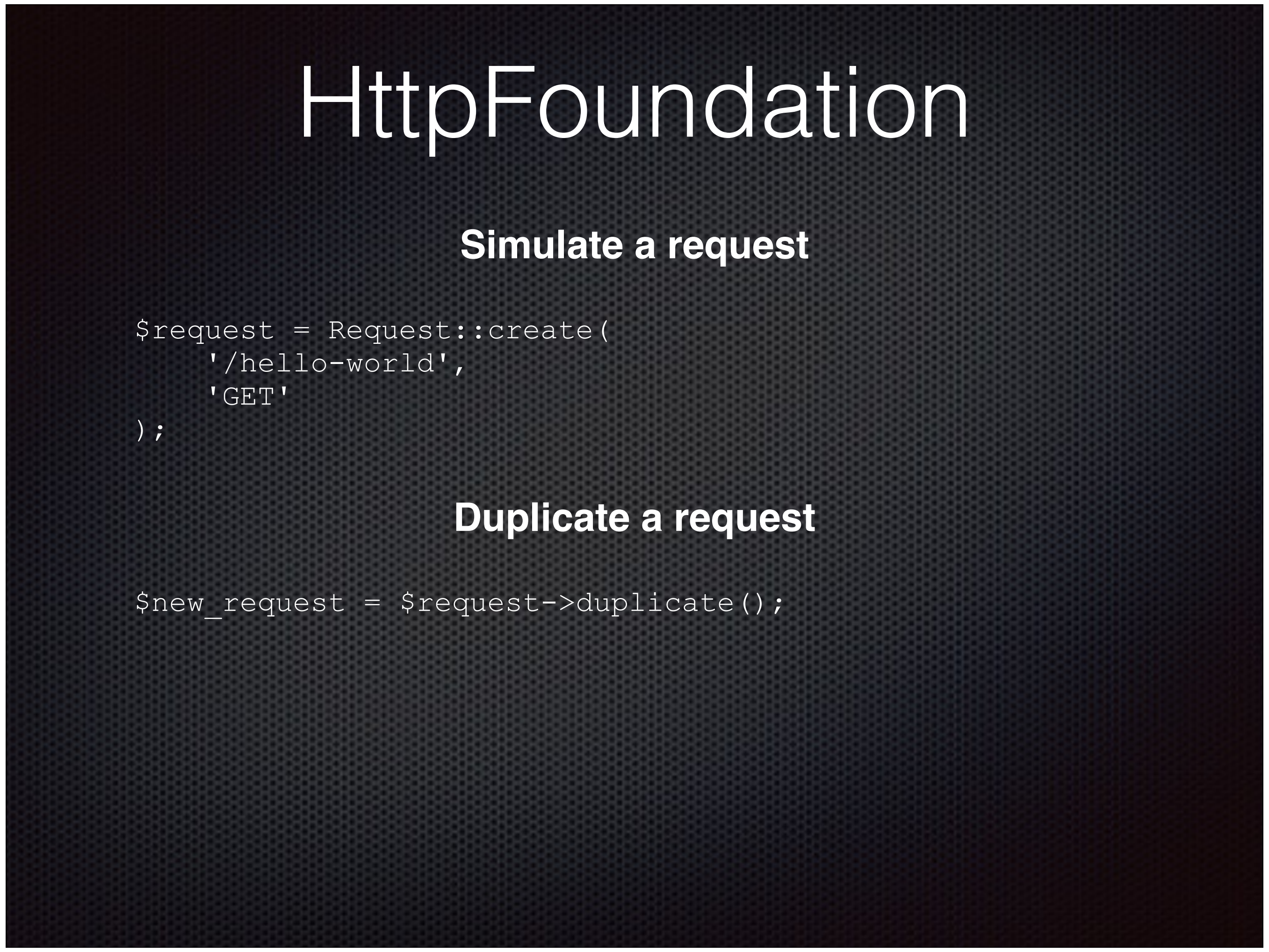


HttpFoundation

Request

- all() Returns the parameters.
- keys() Returns the parameter keys.
- replace() Replaces the current parameters by a new set.
- add() Adds parameters.
- get() Returns a parameter by name.
- set() Sets a parameter by name.
- has() Returns true if the parameter is defined.
- remove() Removes a parameter.
- getAlpha() Returns the alphabetic characters of the parameter value;
- getAlnum() Returns the alphabetic characters and digits of the parameter value;
- getBoolean() Returns the parameter value converted to boolean;
- getDigits() Returns the digits of the parameter value;
- getInt() Returns the parameter value converted to integer;
- filter() Filters the parameter by using the PHP filter_var function.

The Request object and all Request properties are instances or subclasses of Symfony’s ParameterBag class, which means you can also use any of the ParameterBag methods as well, on the Request or any of its properties.


HttpFoundation

Simulate a request

```
$request = Request::create(
    '/hello-world',
    'GET'
);
```

Duplicate a request

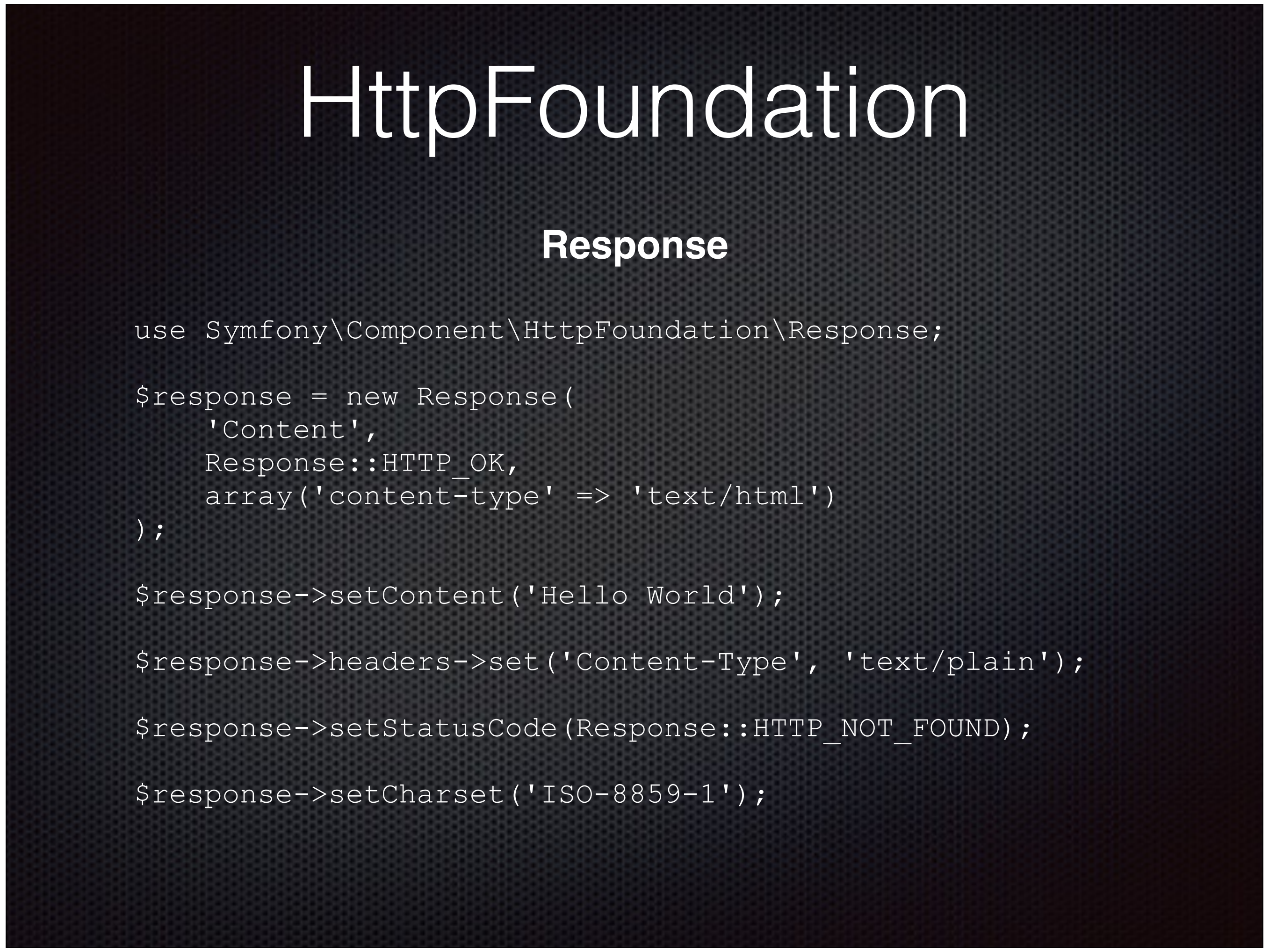
```
$new_request = $request->duplicate();
```

You can simulate a Request object of your own using Request::create().

You can also duplicate an existing request with the duplicate() method.

See why you want to be using that request_stack service?

Now that we’ve dealt with the incoming…



HttpFoundation

Response

```
use Symfony\Component\HttpFoundation\Response;

$response = new Response(
    'Content',
    Response::HTTP_OK,
    array('content-type' => 'text/html')
);

$response->setContent('Hello World');

$response->headers->set('Content-Type', 'text/plain');

$response->setStatusCode(Response::HTTP_NOT_FOUND);

$response->setCharset('ISO-8859-1');
```

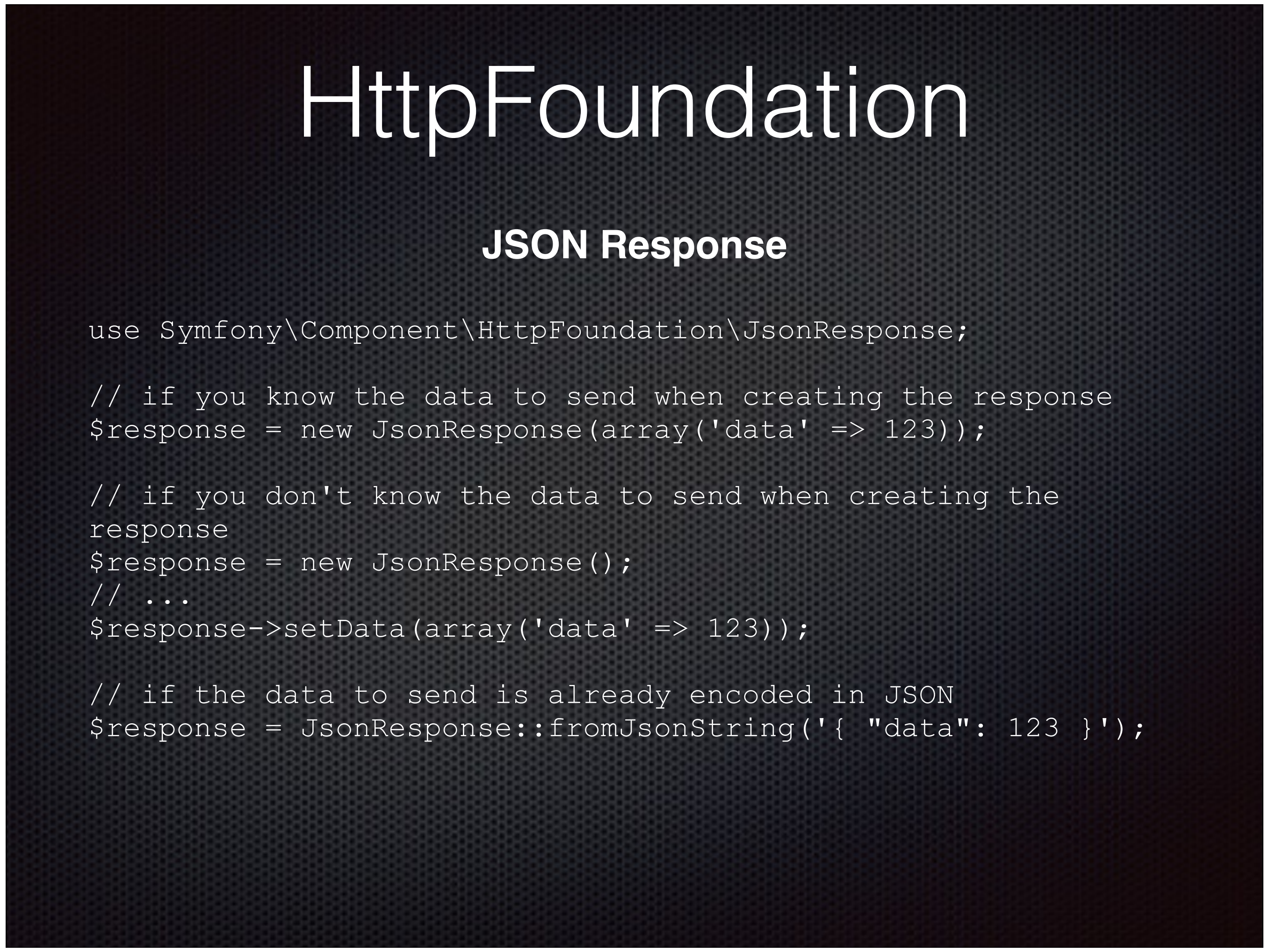
…let’s take a look at the return.

A successful Symfony transaction results in a valid Symfony response, and HttpFoundation provides the Response class for that.

Using this class, you create your response, set any attributes you want on it and add content to it.

In most cases, we let Drupal take care of this, and just pass along render arrays. But there are also a lot of instances where you don’t need a nicely rendered response, like if you’re generating an XML feed or returning a JSON object in an API call.

Symfony even provides some additional response classes to help with that.



HttpFoundation

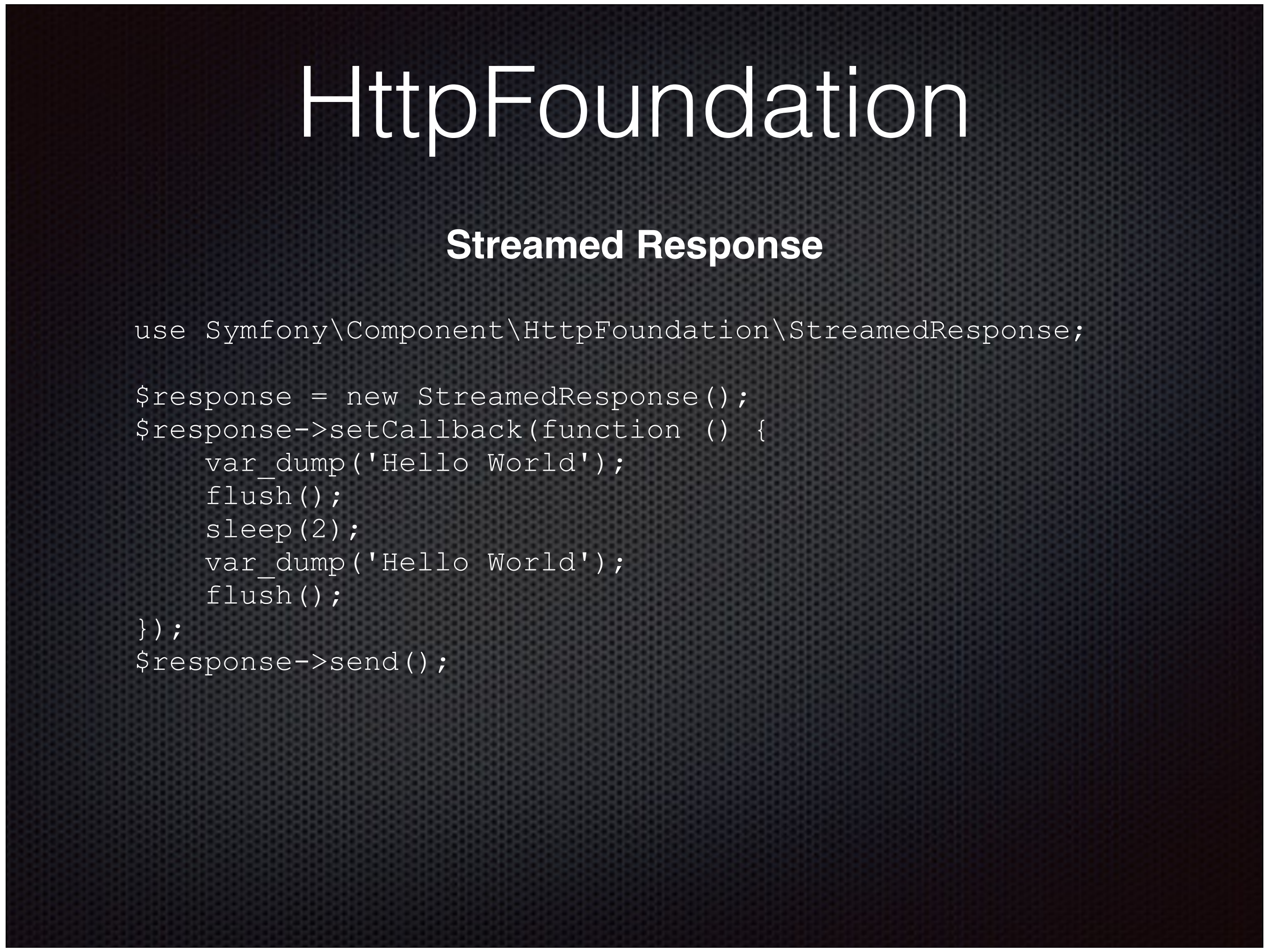
JSON Response

```
use Symfony\Component\HttpFoundation\JsonResponse;

// if you know the data to send when creating the response
$response = new JsonResponse(array('data' => 123));

// if you don't know the data to send when creating the
response
$response = new JsonResponse();
// ...
$response->setData(array('data' => 123));

// if the data to send is already encoded in JSON
$response = JsonResponse::fromJsonString('{ "data": 123 }');
```

If you need a JSON response, use the JsonResponse class.

You can either pass your data to the JsonResponse constructor or instantiate an empty constructor and then set data on that.

JsonResponse takes care of all the nasty details like setting the Content-type header and encoding the data for you.



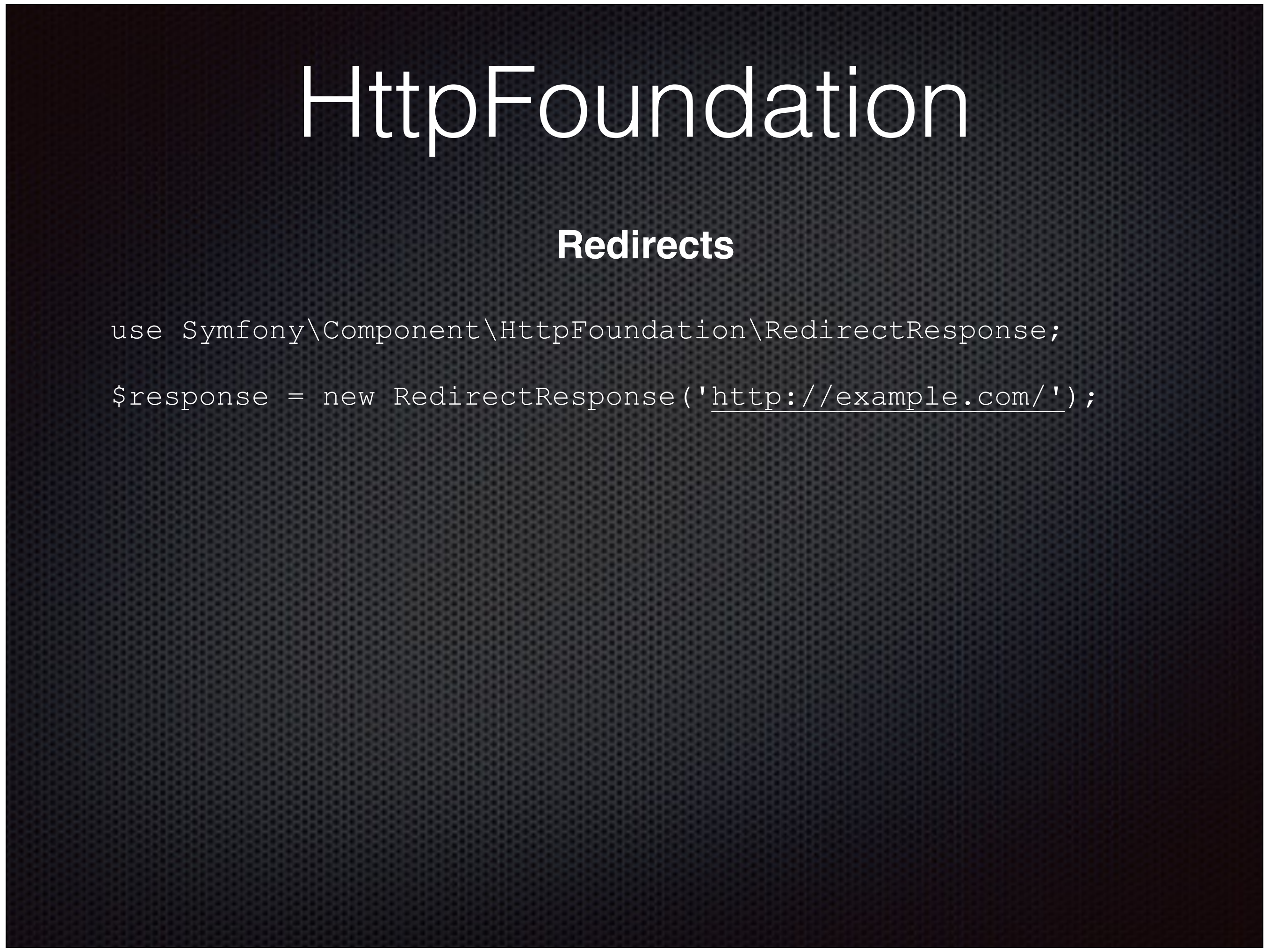
HttpFoundation

Streamed Response

```
use Symfony\Component\HttpFoundation\StreamedResponse;

$response = new StreamedResponse();
$response->setCallback(function () {
    var_dump('Hello World');
    flush();
    sleep(2);
    var_dump('Hello World');
    flush();
});
$response->send();
```

If you’re working with media or really large data transfers, there’s the StreamedResponse.

In this case, the response content is represented by an anonymous callback function instead of a string.

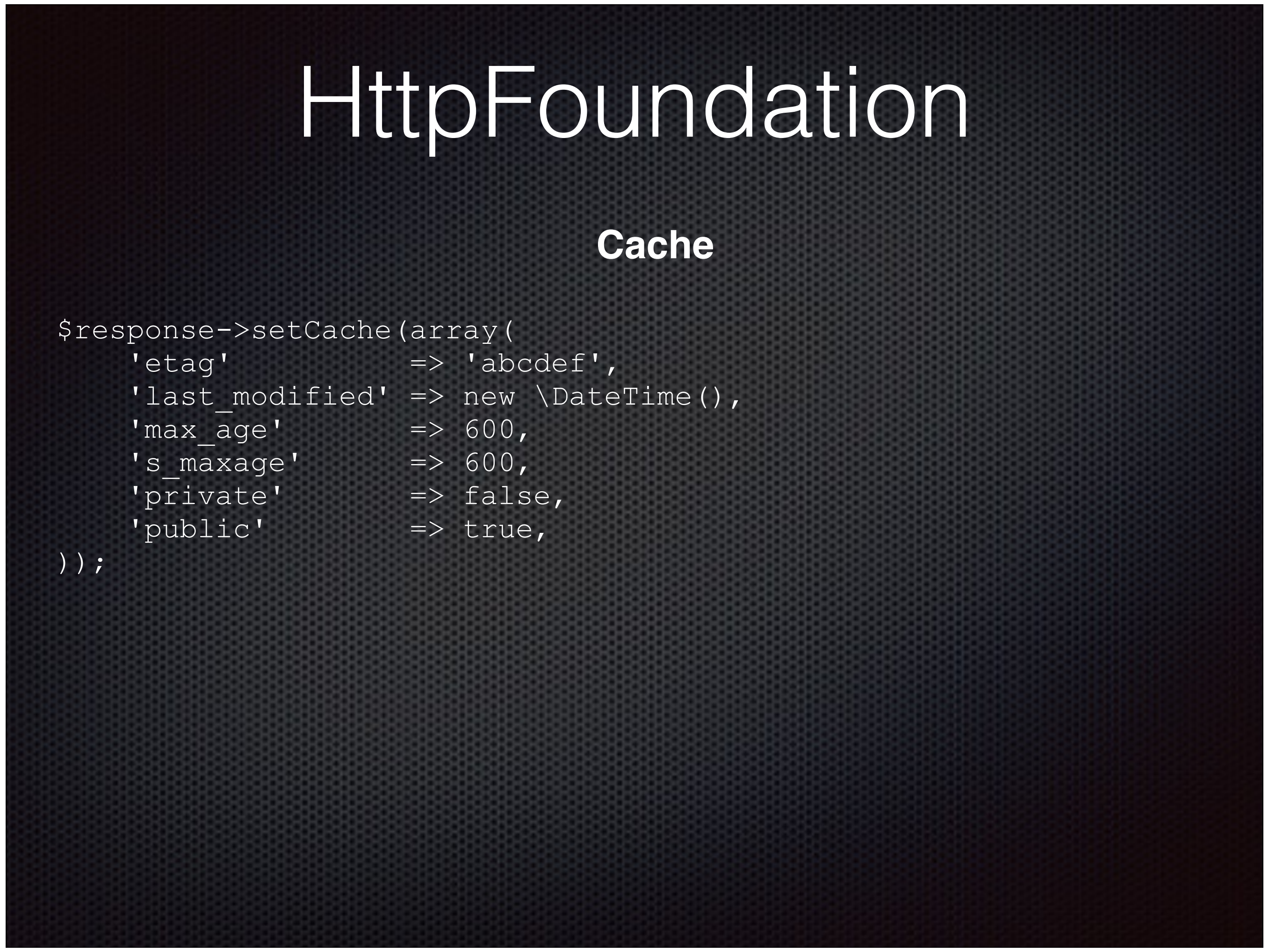


HttpFoundation

Redirects

```
use Symfony\Component\HttpFoundation\RedirectResponse;

$response = new RedirectResponse('http://example.com/');
```

If all you need is a redirect, you can use a RedirectResponse.

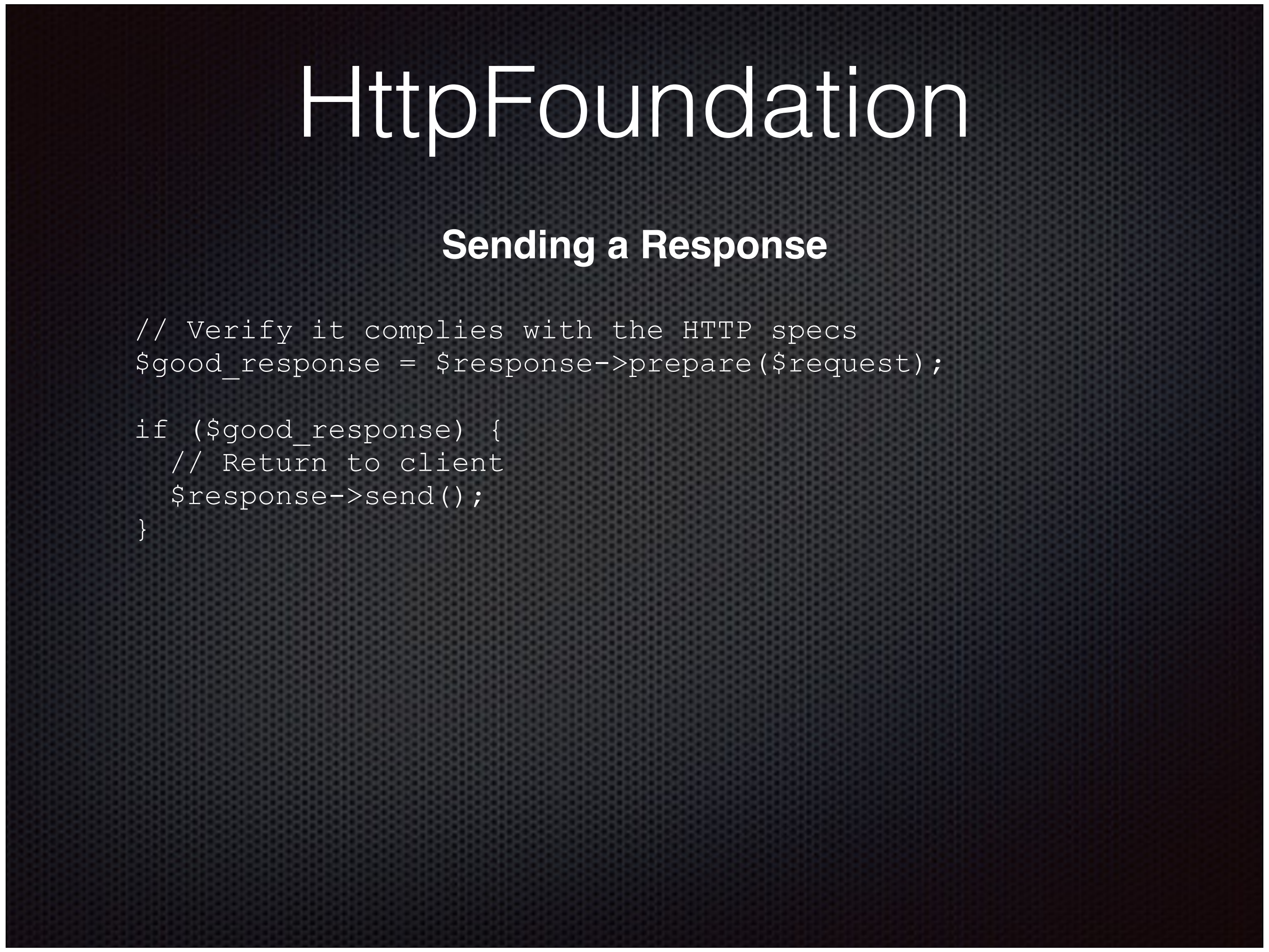


HttpFoundation

Cache

```
$response->setCache(array(
    'etag'          => 'abcdef',
    'last_modified' => new \DateTime(),
    'max_age'       => 600,
    's_maxage'      => 600,
    'private'       => false,
    'public'        => true,
));
```

The Response class also has a setCache() method so you can set caching attributes on your response.

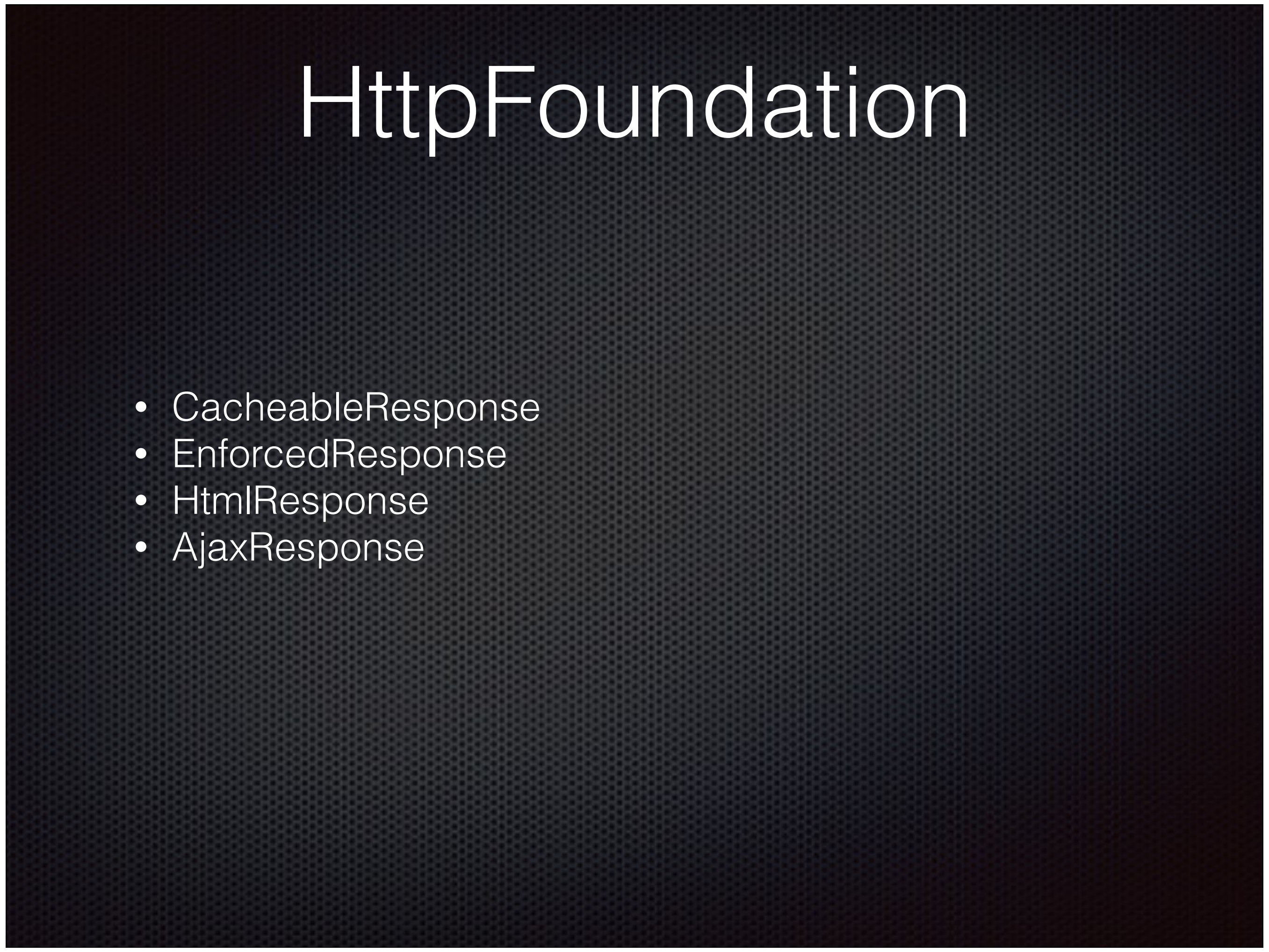

HttpFoundation

Sending a Response

```
// Verify it complies with the HTTP specs
$good_response = $response->prepare($request);

if ($good_response) {
  // Return to client
  $response->send();
}
```

However you've created it, the next step is to call prepare() on your response to make sure it's good.

And if it checks out OK, send it along.



HttpFoundation

- CacheableResponse
- EnforcedResponse
- HtmlResponse
- AjaxResponse

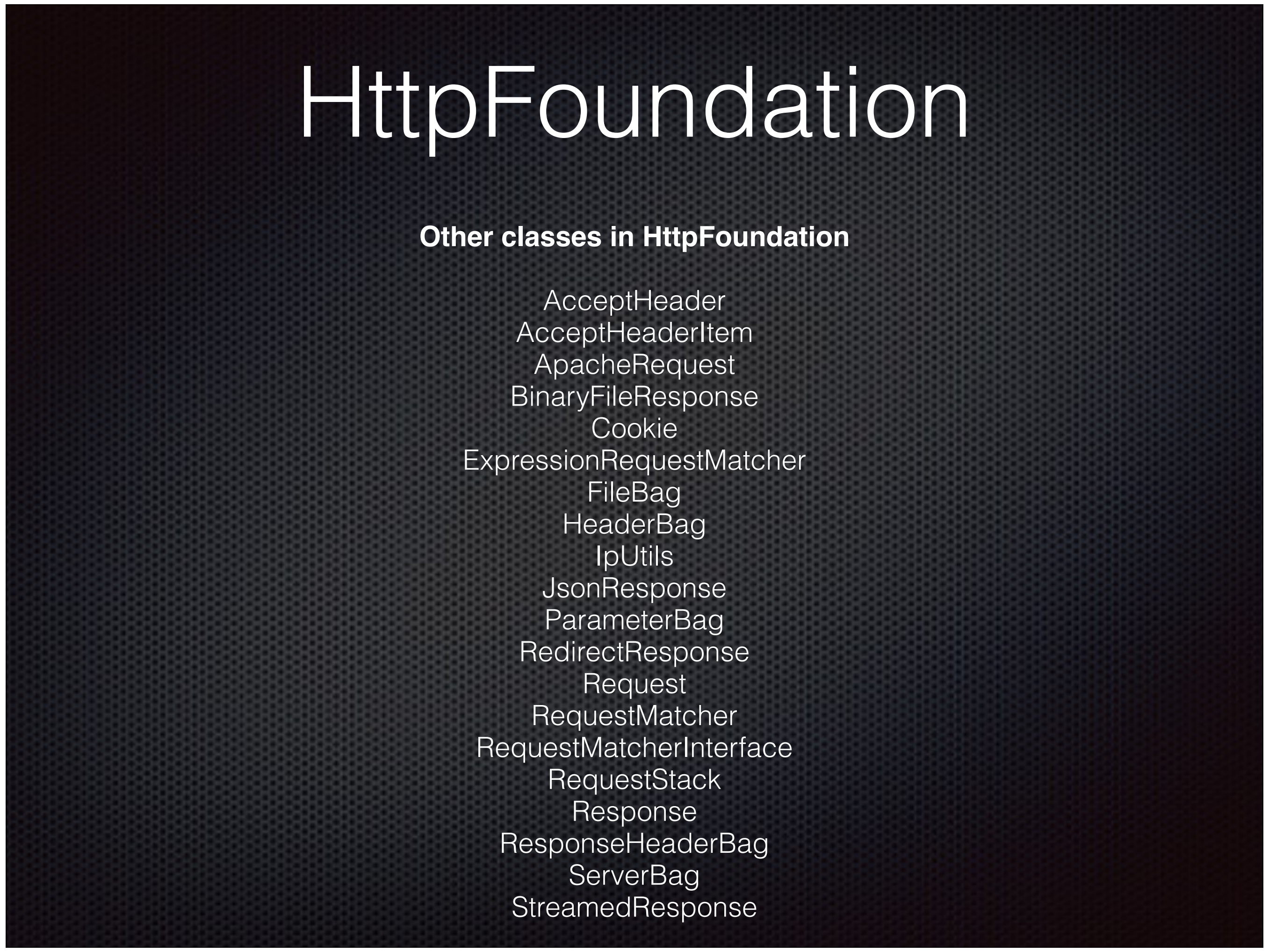
Drupal extends Symfony's response with a few classes of its own.

CacheableResponse exposes cache metadata. Using this one, you can set cache values at the render array level.

EnforcedResponse lets you deal with EnforcedResponseExceptions thrown by Drupal's FormBuilder.

HtmlResponse gives you a CacheableResponse and adds Drupal attached metadata like libraries, settings, http_header and html_head.

AjaxResponse extends JsonResponse to let you add callback commands to take care of Ajax actions.

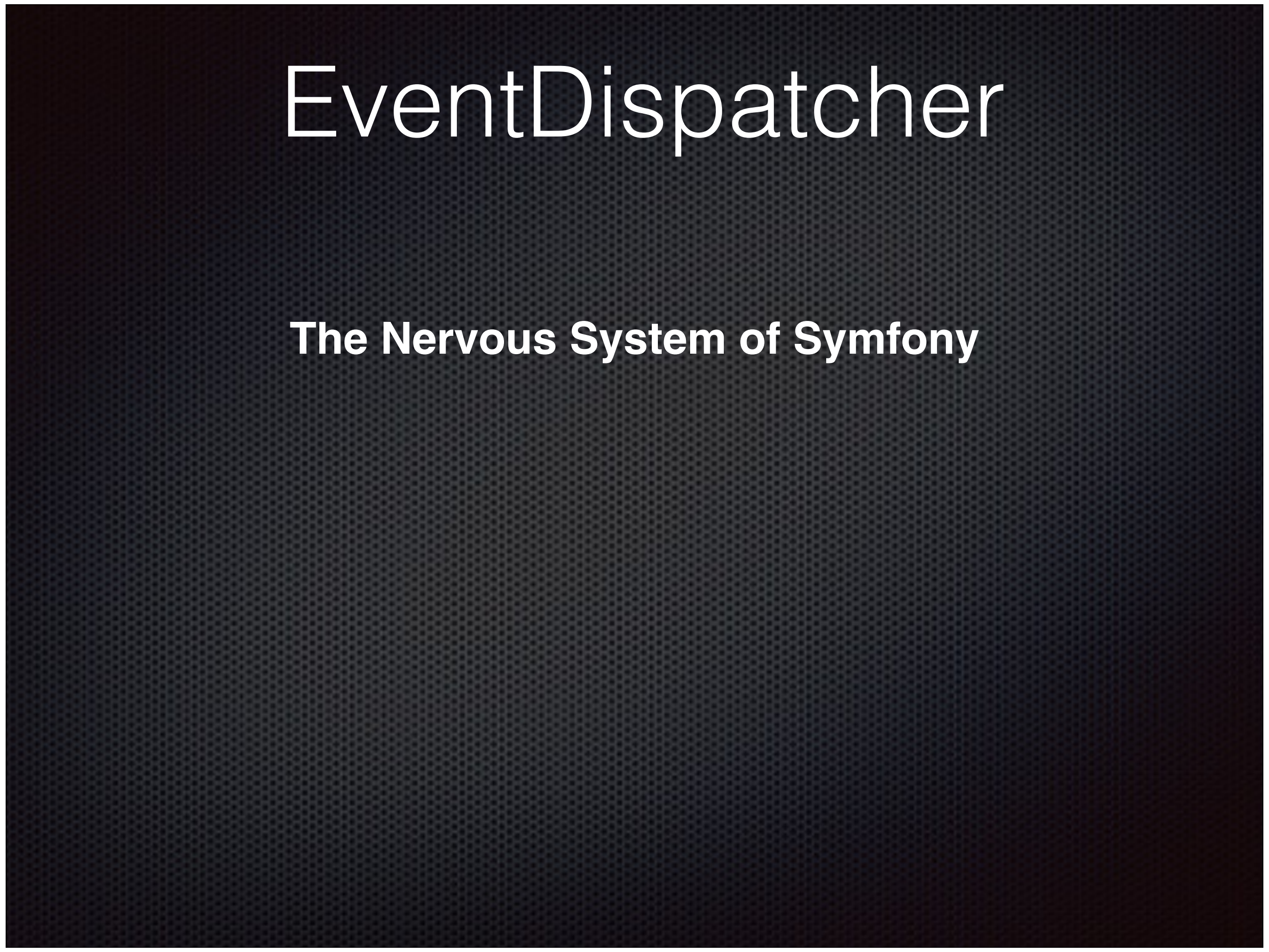


HttpFoundation

Other classes in HttpFoundation

AcceptHeader
AcceptHeaderItem
ApacheRequest
BinaryFileResponse
Cookie
ExpressionRequestMatcher
FileBag
HeaderBag
IpUtils
JsonResponse
ParameterBag
RedirectResponse
Request
RequestMatcher
RequestMatcherInterface
RequestStack
Response
ResponseHeaderBag
ServerBag
StreamedResponse

HttpFoundation itself has lots of other classes, too. But we've got other things to talk about, so all you get is this quick list before we move on to…



EventDispatcher

The Nervous System of Symfony

…the EventDispatcher component. This is one that really changes things for us.

How many of you are coming to Drupal 8 from Drupal 7?

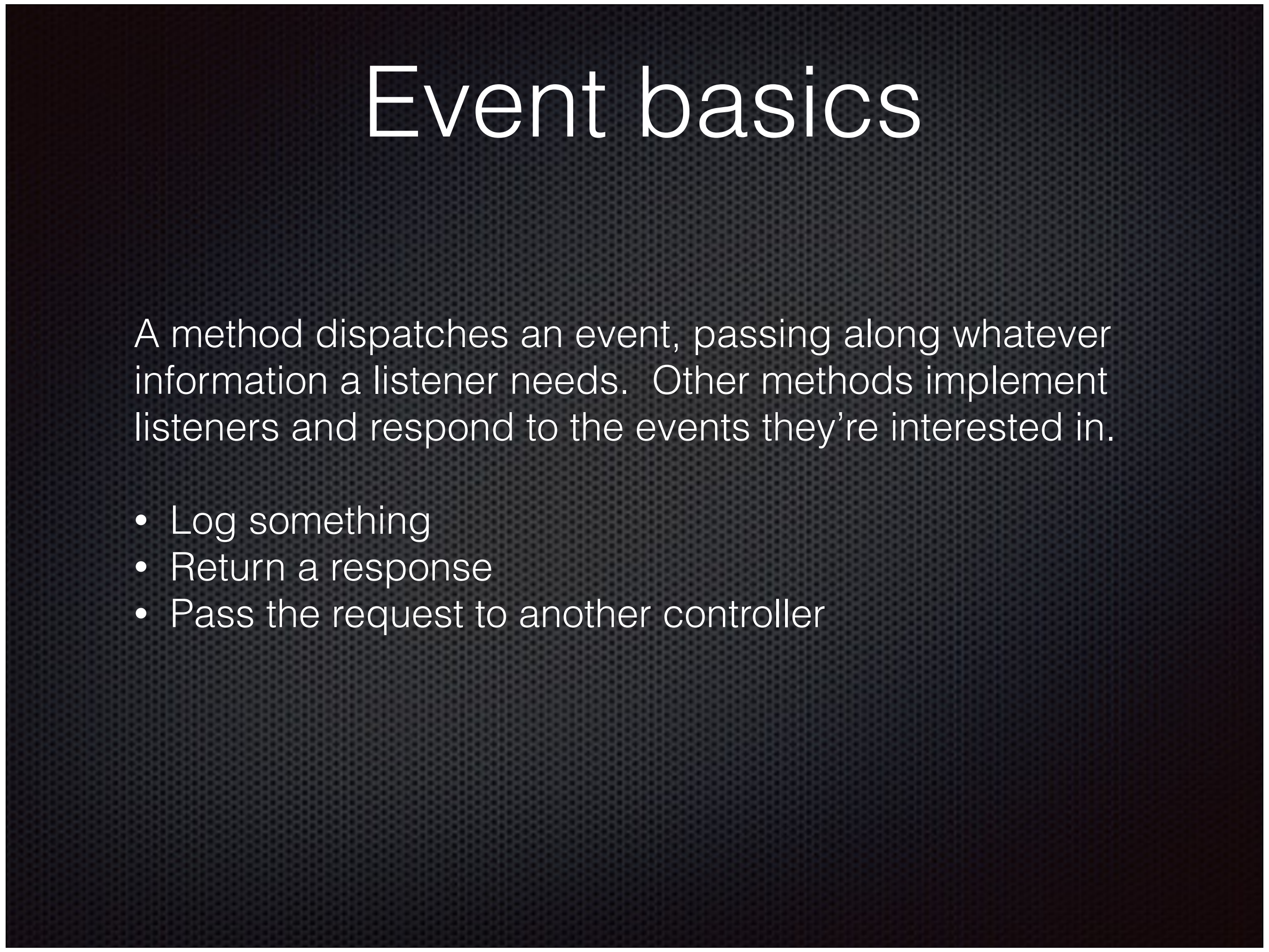
How many of you are getting into Drupal as of D8?

Up through D7, Drupal relied on the hook system. Core exposed various methods as hooks that modules could implement themselves to affect the request/response cycle.

A big part of implementing a module was figuring out what hook to use to tap into the process at the appropriate time.

Moving from one version of Drupal to the next was a matter of learning what hooks had been added or deprecated and modifying module code accordingly.

Drupal 8 still has hooks, but Symfony also give us the EventDispatcher, which offers a lot more flexibility.


Event basics

A method dispatches an event, passing along whatever information a listener needs. Other methods implement listeners and respond to the events they're interested in.

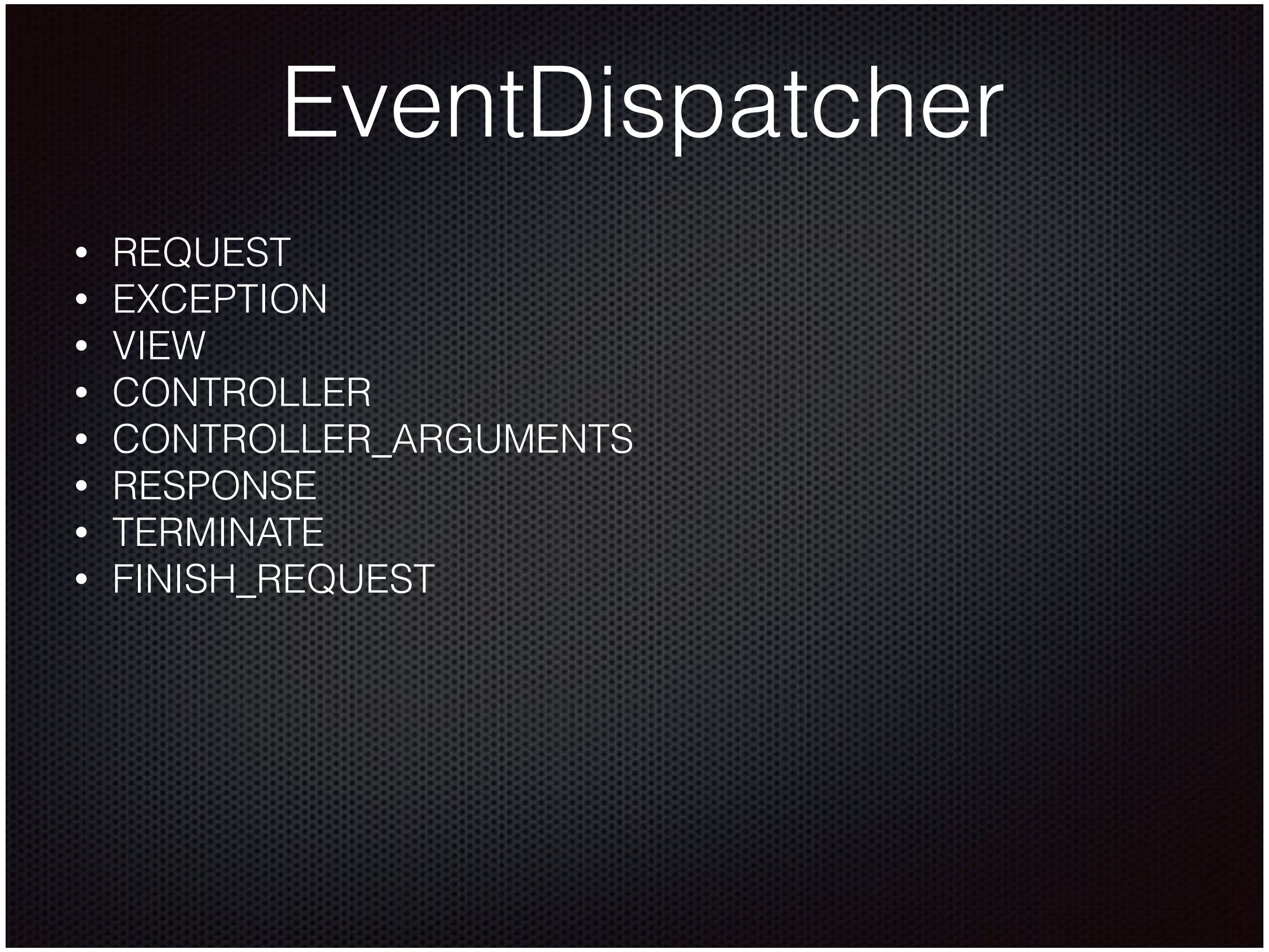
- Log something
- Return a response
- Pass the request to another controller

With EventDispatcher, code can dispatch an event at any time.

Other code can listen for events and respond immediately when an event occurs that it know about.

The dispatcher dispatches an event and checks to see if anyone responds. It then processes any received responses and then moves on.

This means that, rather than being stuck with affecting processing when Drupal allows it, we can send and respond to events whenever we want.



EventDispatcher

- REQUEST
- EXCEPTION
- VIEW
- CONTROLLER
- CONTROLLER_ARGUMENTS
- RESPONSE
- TERMINATE
- FINISH_REQUEST

Symfony defines 8 events.

The Request event happens at the very beginning of request dispatching. This allows you to modify the request or create a response before any other code is executed.

The Exception event happens when an uncaught exception shows up so you can create a response or modify the thrown exception

The View event happens when the return value of a controller isn’t a valid HttpFoundation Response object. It lets you create a response

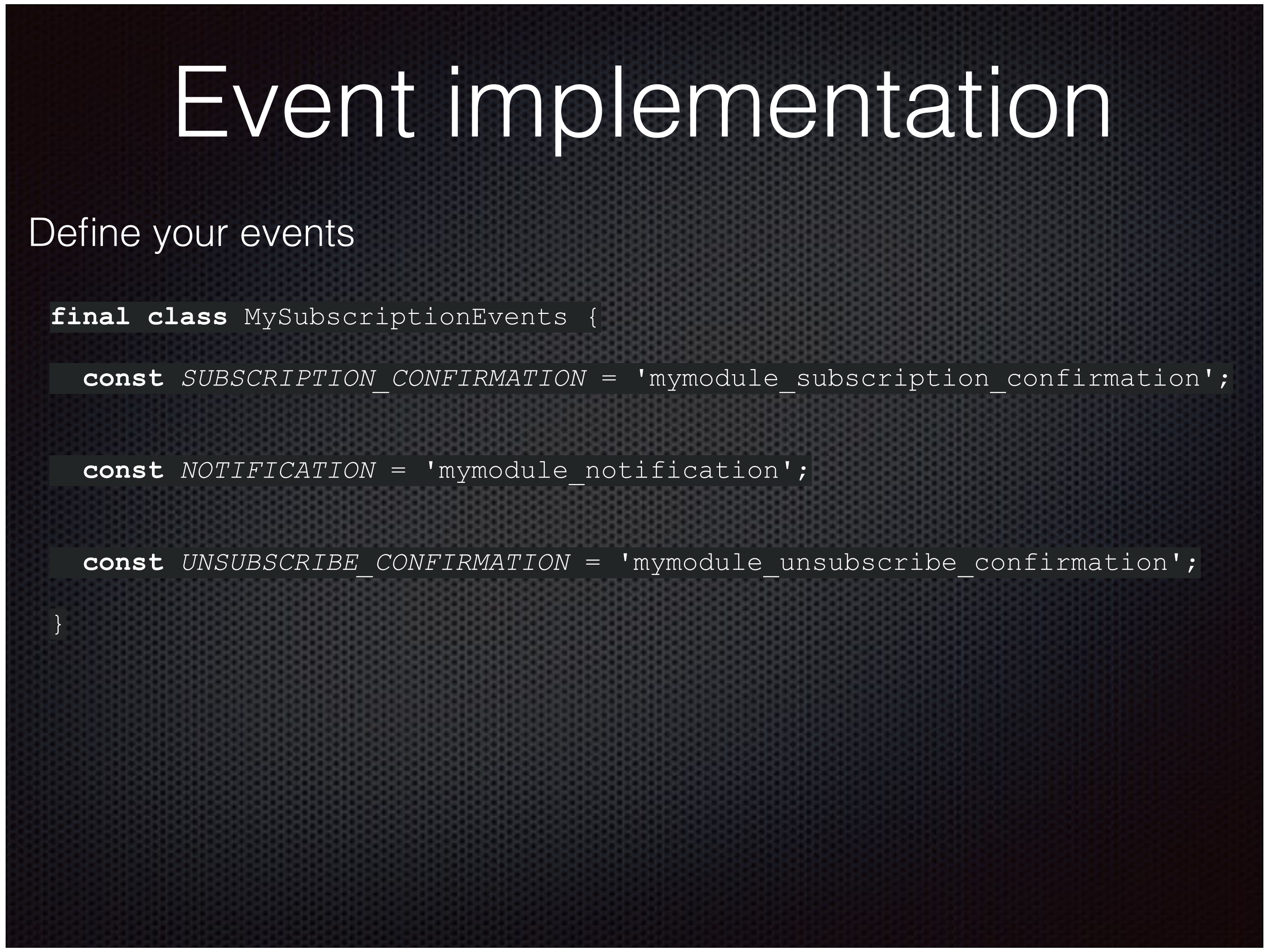
The Controller event happens when a controller is found that handles the current request. It allows you to specify a different controller for the request.

Controller Arguments occurs when controller arguments have been resolved and lets you change the arguments that will be passed fo the controller.

The Response event happens when a response has been created. This gives you an opportunity to modify or replace the response that will be returned.

The Terminate event happens after the response has been sent. This one lets lets you run expensive jobs without delaying the response.

The Finish Request occurs when a response has been generated and is useful for resetting application state, if it was changed during the request.



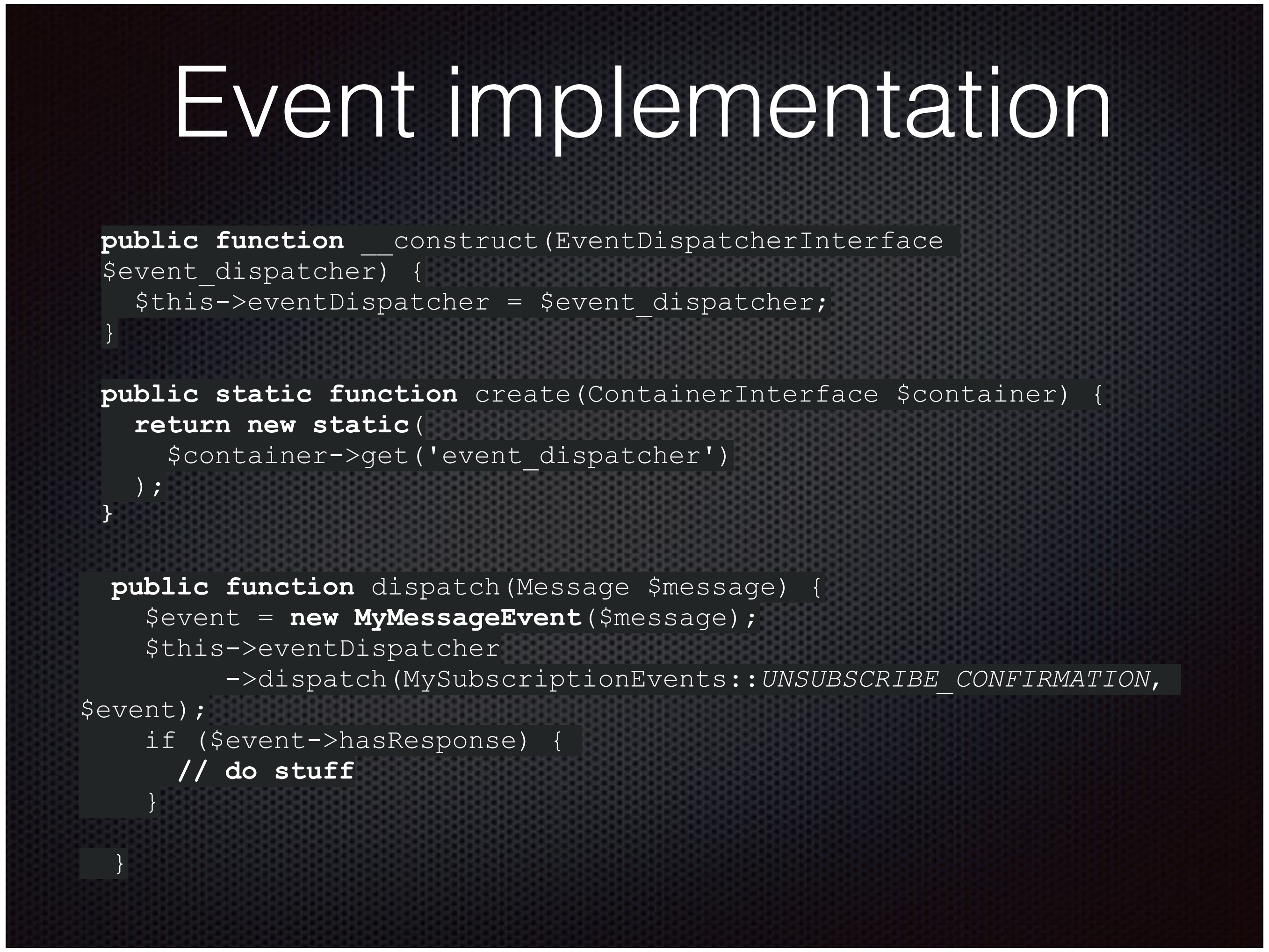
Event implementation

Define your events

```
final class MySubscriptionEvents {

  const SUBSCRIPTION_CONFIRMATION = 'mymodule_subscription_confirmation';

  const NOTIFICATION = 'mymodule_notification';

  const UNSUBSCRIBE_CONFIRMATION = 'mymodule_unsubscribe_confirmation';

}
```

Symfony also makes it very easy to create your own events. Lots of Drupal core modules do this, and so can you.

Step 1: Create a class that defines the events you want to handle. Note that these are just consants.

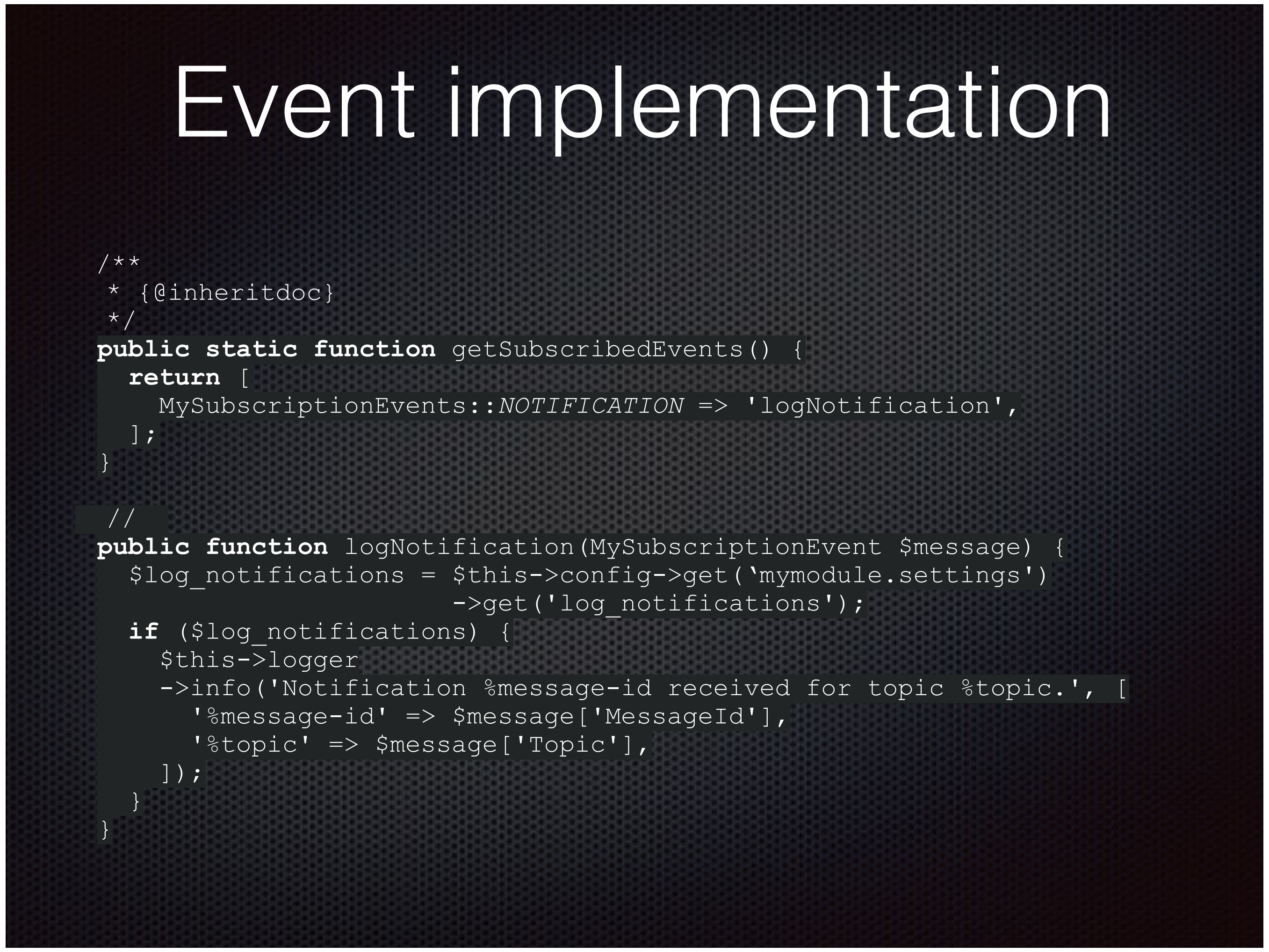


Event implementation

```
 public function __construct(EventDispatcherInterface
 $event_dispatcher) {
   $this->eventDispatcher = $event_dispatcher;
 }

 public static function create(ContainerInterface $container) {
   return new static(
     $container->get('event_dispatcher')
   );
 }


  public function dispatch(Message $message) {
    $event = new MyMessageEvent($message);
    $this->eventDispatcher
         ->dispatch(MySubscriptionEvents::UNSUBSCRIBE_CONFIRMATION,
$event);
    if ($event->hasResponse) {
      // do stuff
    }

  }
```

Step 2: You need something to dispatch events, so you create a class using event_dispatcher service.

This class implement the dispatch() method that takes whatever data you might want to pass along to your listeners, instantiates an event on it, and calls Symfony EventDispatcher's dispatch() on the resulting event. If the event gets a response, do something with it. You might log the event or return the response. You could even dispatch another event. And on and on.



Event implementation

```
/**
 * {@inheritdoc}
 */
public static function getSubscribedEvents() {
  return [
    MySubscriptionEvents::NOTIFICATION => 'logNotification',
  ];
}

 //
public function logNotification(MySubscriptionEvent $message) {
  $log_notifications = $this->config->get('mymodule.settings')
                        ->get('log_notifications');
  if ($log_notifications) {
    $this->logger
    ->info('Notification %message-id received for topic %topic.', [
      '%message-id' => $message['MessageId'],
      '%topic' => $message['Topic'],
    ]);
  }
}
```

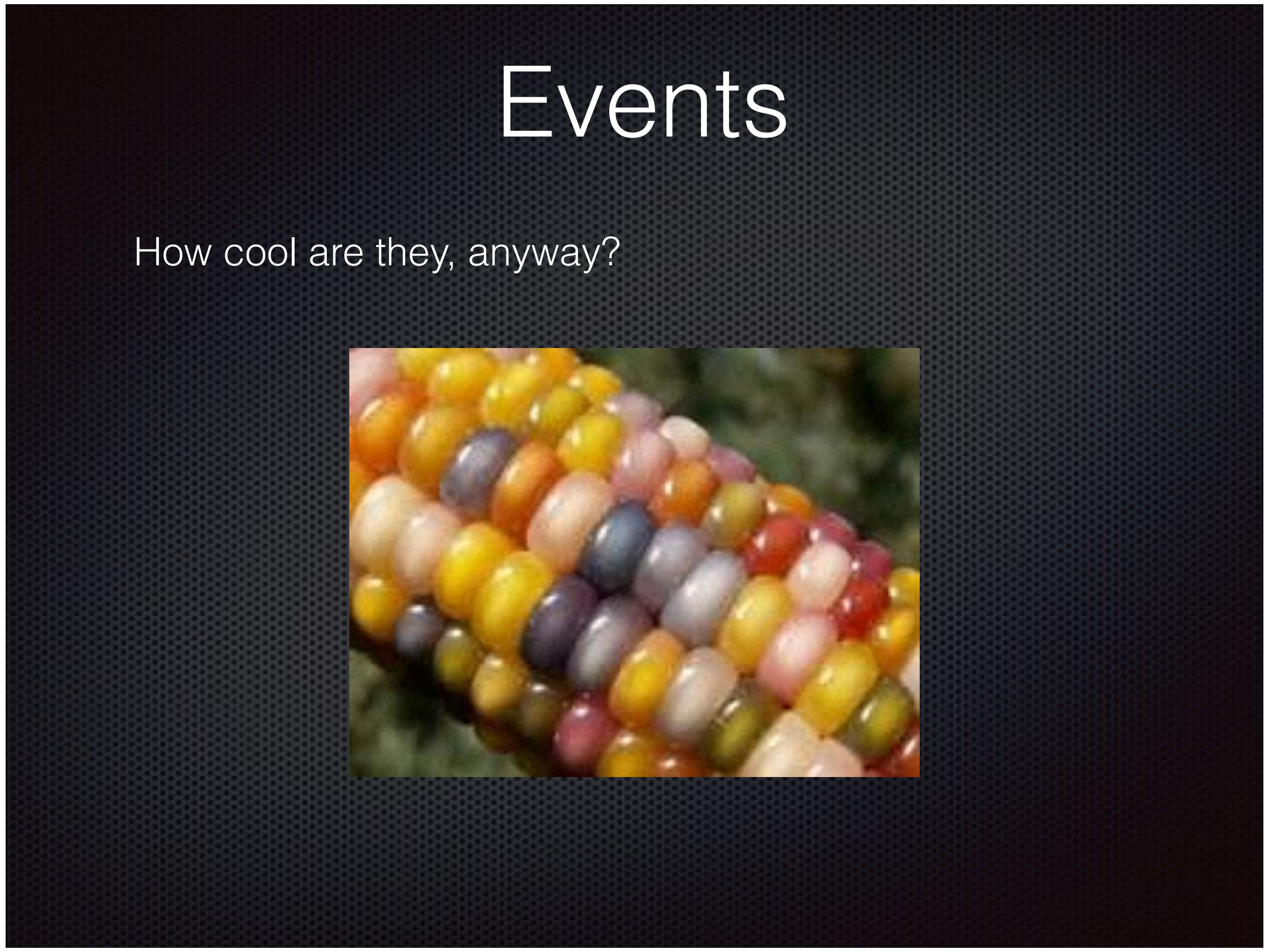
Step 3
To respond to events, create a subscriber class. This class must implement Symfony\Component\EventDispatcher\EventSubscriberInterface

To satisfy the interface, implement getSubscribedEvents().

Here, you specify what event(s) you want to process and specify the callback(s) to use for each event.

Define your callback(s) and you're done.

In this case, when we get a notification event, we're going to call the logNotification() method to log the event.

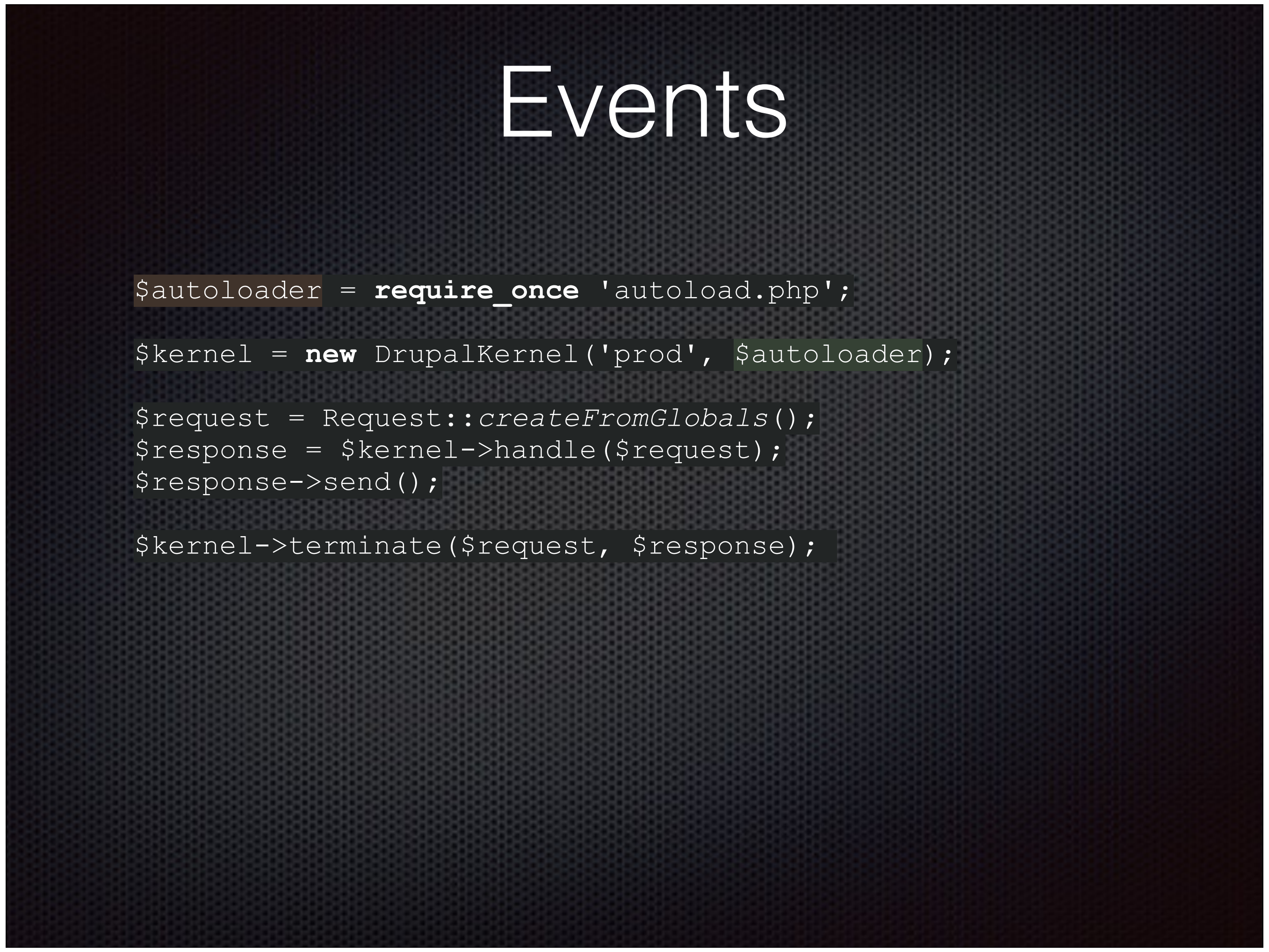


Events

How cool are they, anyway?

So how cool are events?

This picture of some really pretty corn is to get you thinking about kernels.

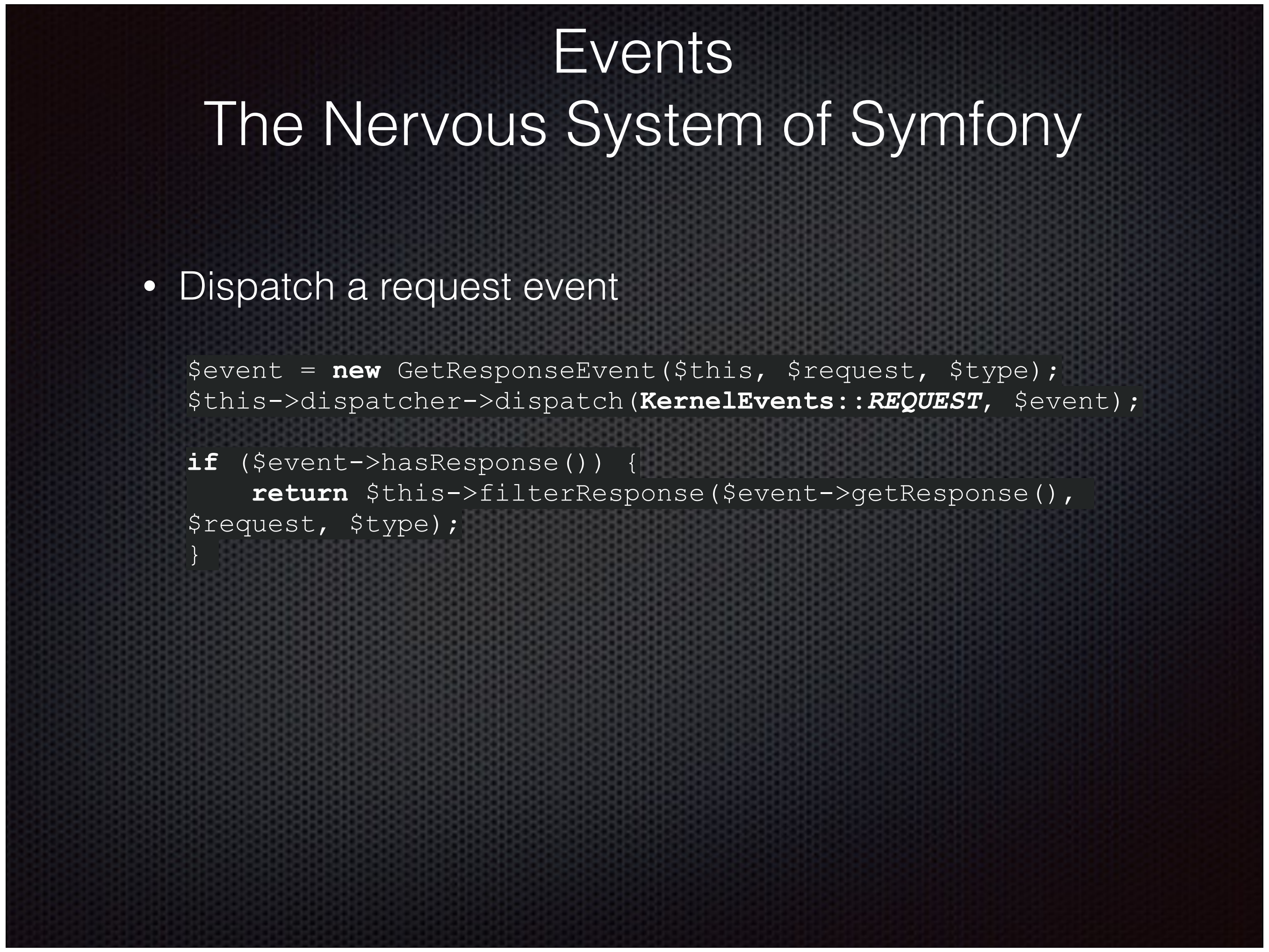

Events

```
$autoloader = require_once 'autoload.php';

$kernel = new DrupalKernel('prod', $autoloader);

$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();

$kernel->terminate($request, $response);
```

With that in mind, let’s take a look at Drupal’s index.php.

It instantiates a DrupalKernel object, gets a Symfony Request object, passes that to the DrupalKernel->handle() method and finally returns some kind of response.

DrupalKernel->handle() calls Symfony’s HttpKernel->handle(), which wraps another method that just dispatches events and hopes for responses.

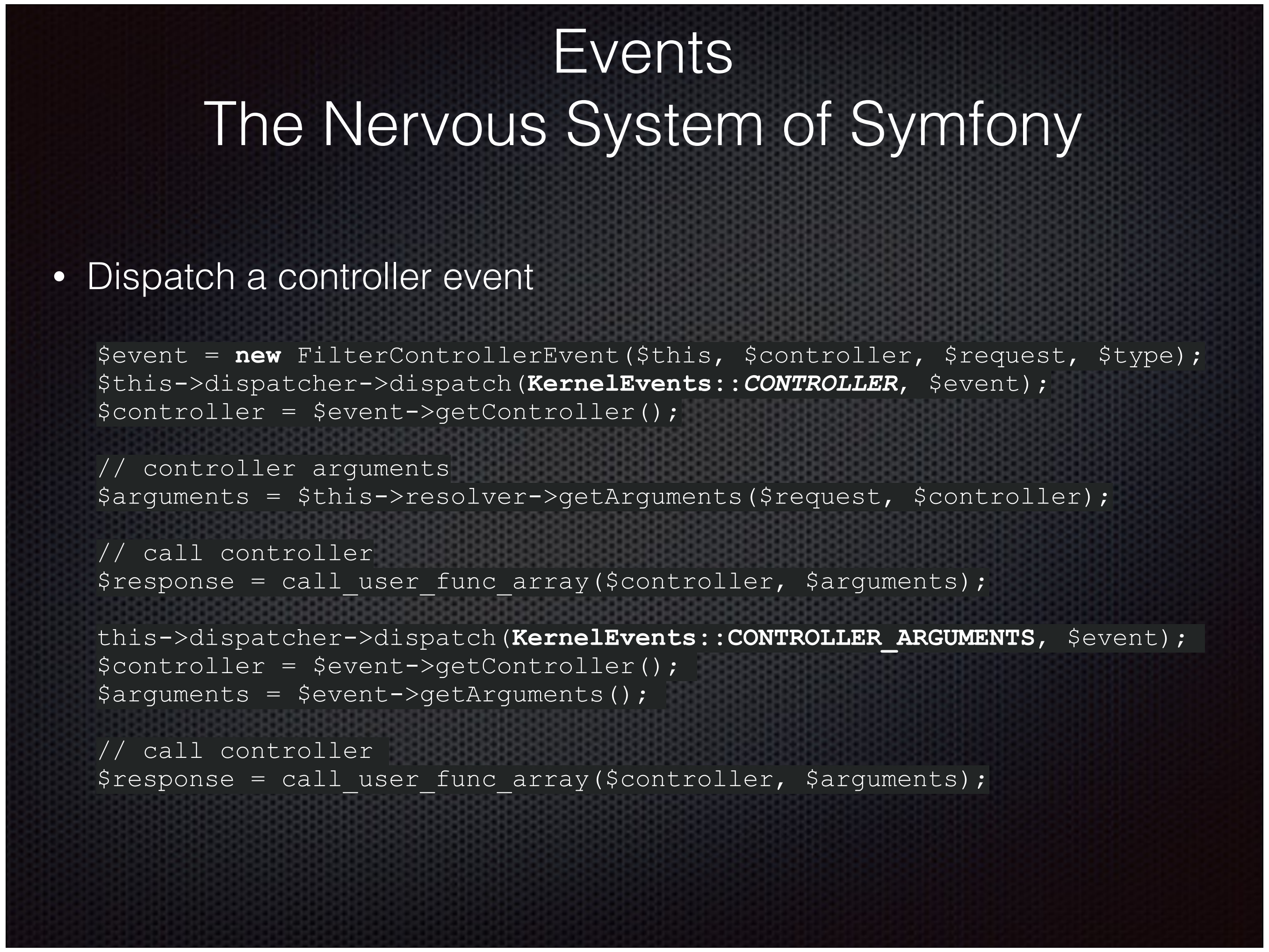


Events
The Nervous System of Symfony

- Dispatch a request event

```
$event = new GetResponseEvent($this, $request, $type);
$this->dispatcher->dispatch(KernelEvents::REQUEST, $event);

if ($event->hasResponse()) {
    return $this->filterResponse($event->getResponse(),
$request, $type);
}
```

First it dispatches a Request event and checks for any one of those valid Response types we talked about earlier. If there's a response, we're done.

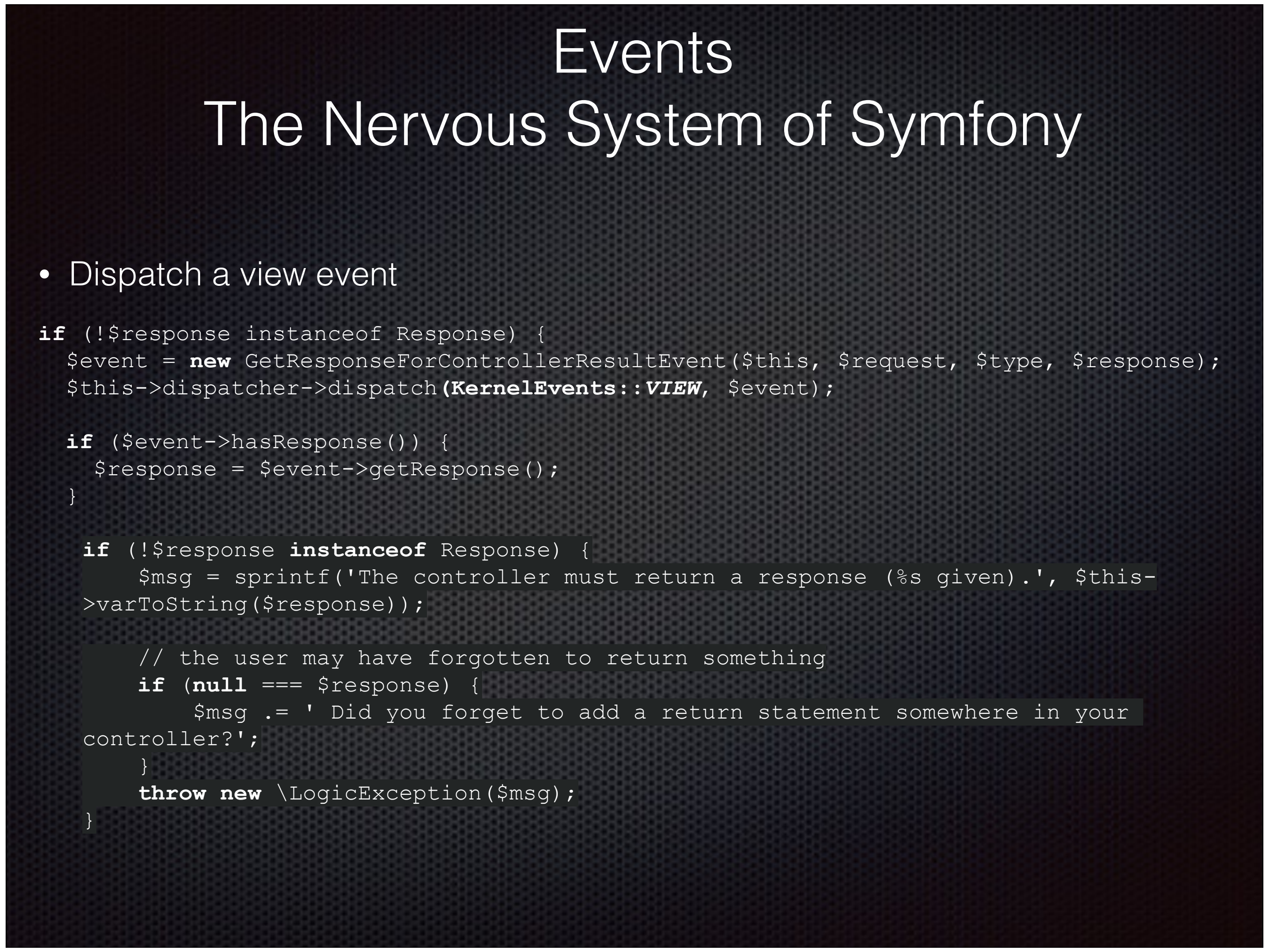


Events
The Nervous System of Symfony

- Dispatch a controller event

```
$event = new FilterControllerEvent($this, $controller, $request, $type);
$this->dispatcher->dispatch(KernelEvents::CONTROLLER, $event);
$controller = $event->getController();

// controller arguments
$arguments = $this->resolver->getArguments($request, $controller);

// call controller
$response = call_user_func_array($controller, $arguments);

this->dispatcher->dispatch(KernelEvents::CONTROLLER_ARGUMENTS, $event);
$controller = $event->getController();
$arguments = $event->getArguments();

// call controller
$response = call_user_func_array($controller, $arguments);
```

If nothing responds to the request event, it dispatches Controller and ControllerArgument events.



Events
The Nervous System of Symfony

- Dispatch a view event

```
if (!$response instanceof Response) {
  $event = new GetResponseForControllerResultEvent($this, $request, $type, $response);
  $this->dispatcher->dispatch(KernelEvents::VIEW, $event);

  if ($event->hasResponse()) {
    $response = $event->getResponse();
  }

   if (!$response instanceof Response) {
       $msg = sprintf('The controller must return a response (%s given).', $this-
   >varToString($response));

       // the user may have forgotten to return something
       if (null === $response) {
           $msg .= ' Did you forget to add a return statement somewhere in your
   controller?';
       }
       throw new \LogicException($msg);
   }
```

If the controller events don’t get a response, handleRaw() dispatches a View event.

And if that doesn’t get a valid response, you get what probably is, or (if you’re new to D8) will certainly become, a fairly familiar error.

If you’ve been paying attention, you might have noticed that you can help out your performance a bit by replacing a controller with a class that listens for Request events.

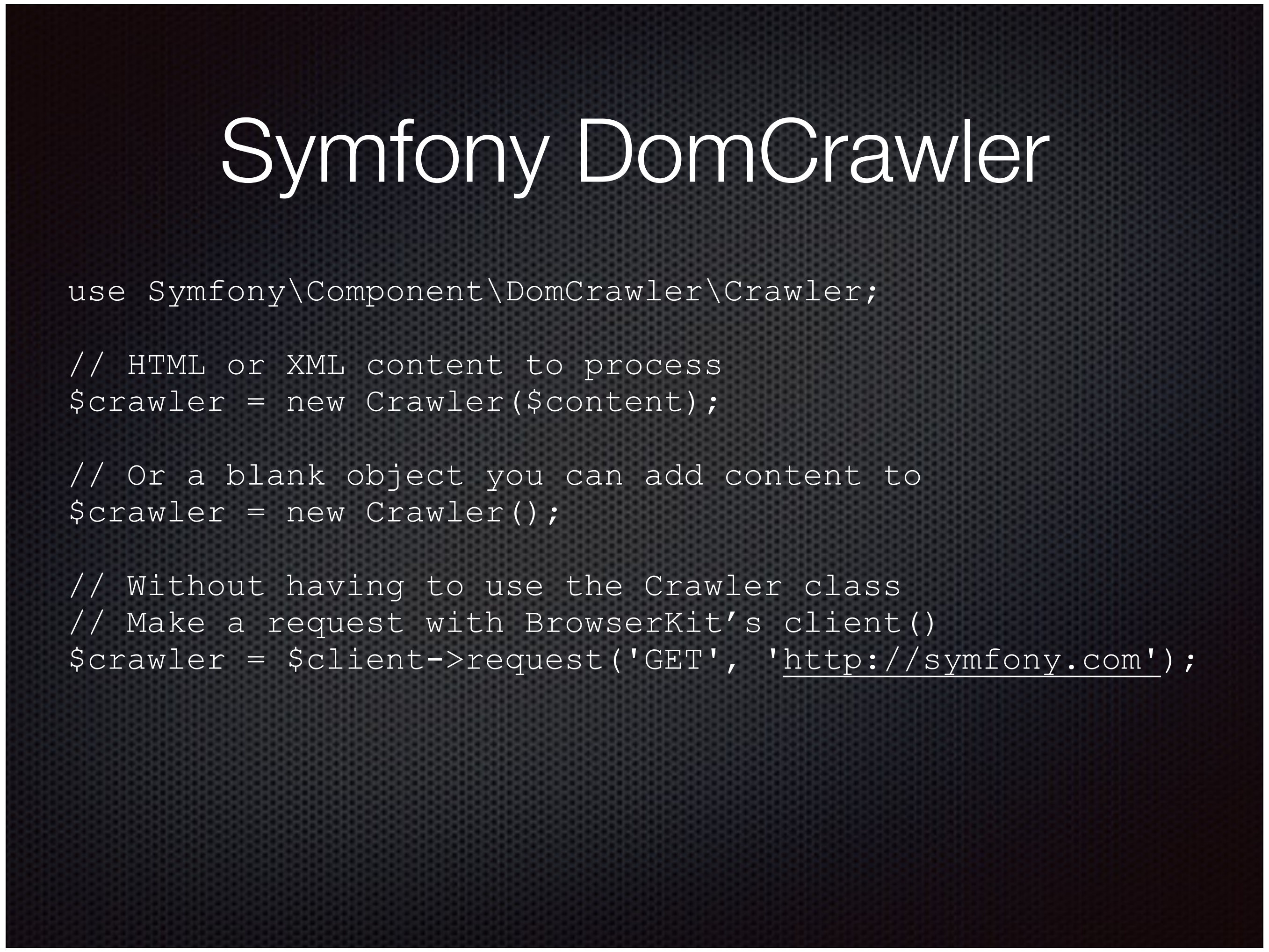
That’s a bit of under-the-hood, now let’s take a look at a few fun extras.

Symfony DomCrawler

XML/HTML

First up is the DomCrawler component. This one lets you navigate/manipulate a DOM, very similar to what jQuery lets you do in javascript.

This one's pretty useful if you're doing any scraping or datamining to get content from other sites. My boss used this to write a module that gets the list of sessions from the Drupalcon site.

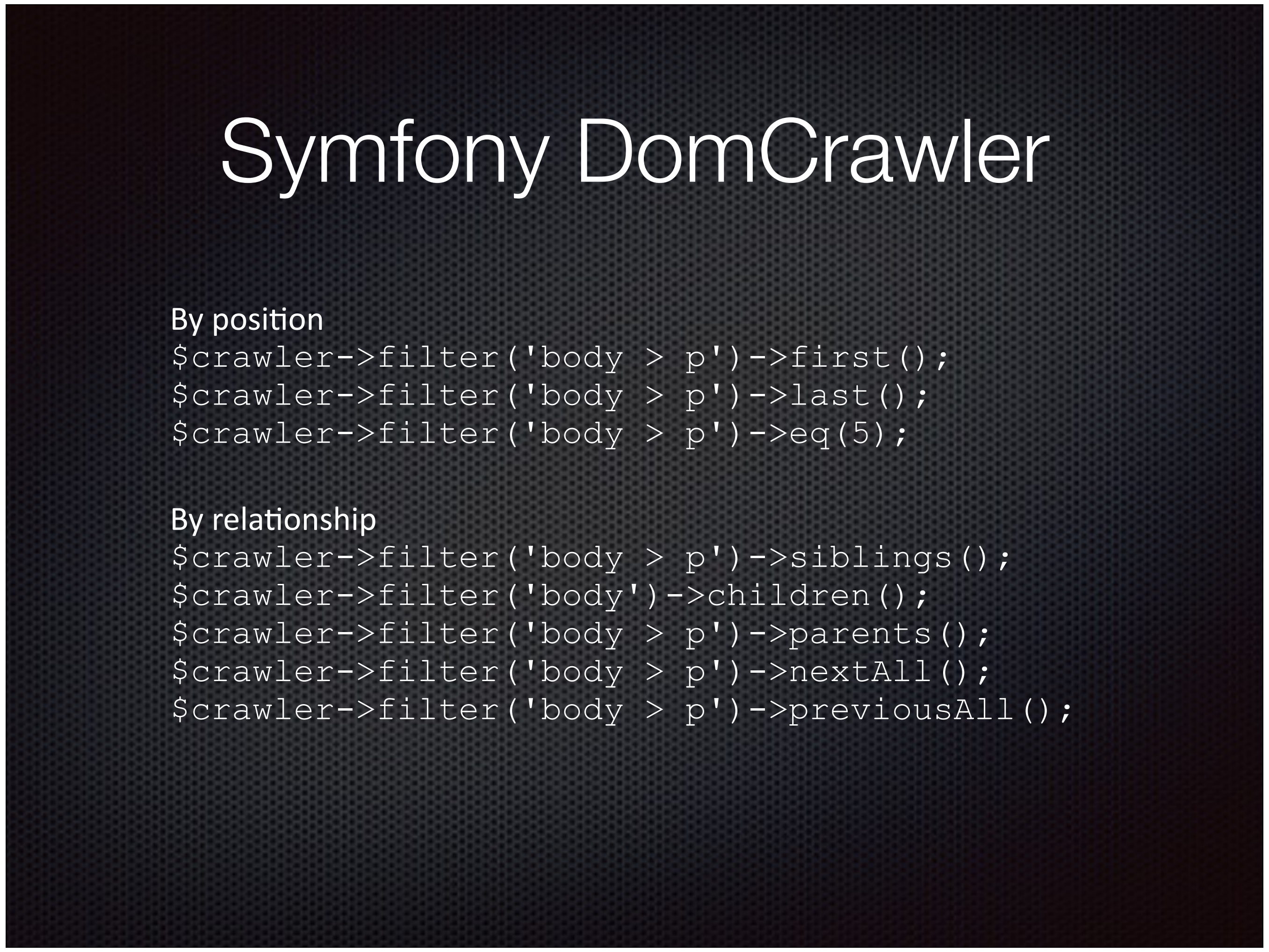

Symfony DomCrawler

```
use Symfony\Component\DomCrawler\Crawler;

// HTML or XML content to process
$crawler = new Crawler($content);

// Or a blank object you can add content to
$crawler = new Crawler();

// Without having to use the Crawler class
// Make a request with BrowserKit's client()
$crawler = $client->request('GET', 'http://symfony.com');
```

To use it, just include Symfony\Component\DomCrawler\Crawler in your code

Instantiate a new crawler with or without content. The latter lets you use the crawler to create and populate a new document.

If you're using the Symfony BrowserKit class to make a client request, the response will already be a Crawler object.



Symfony DomCrawler

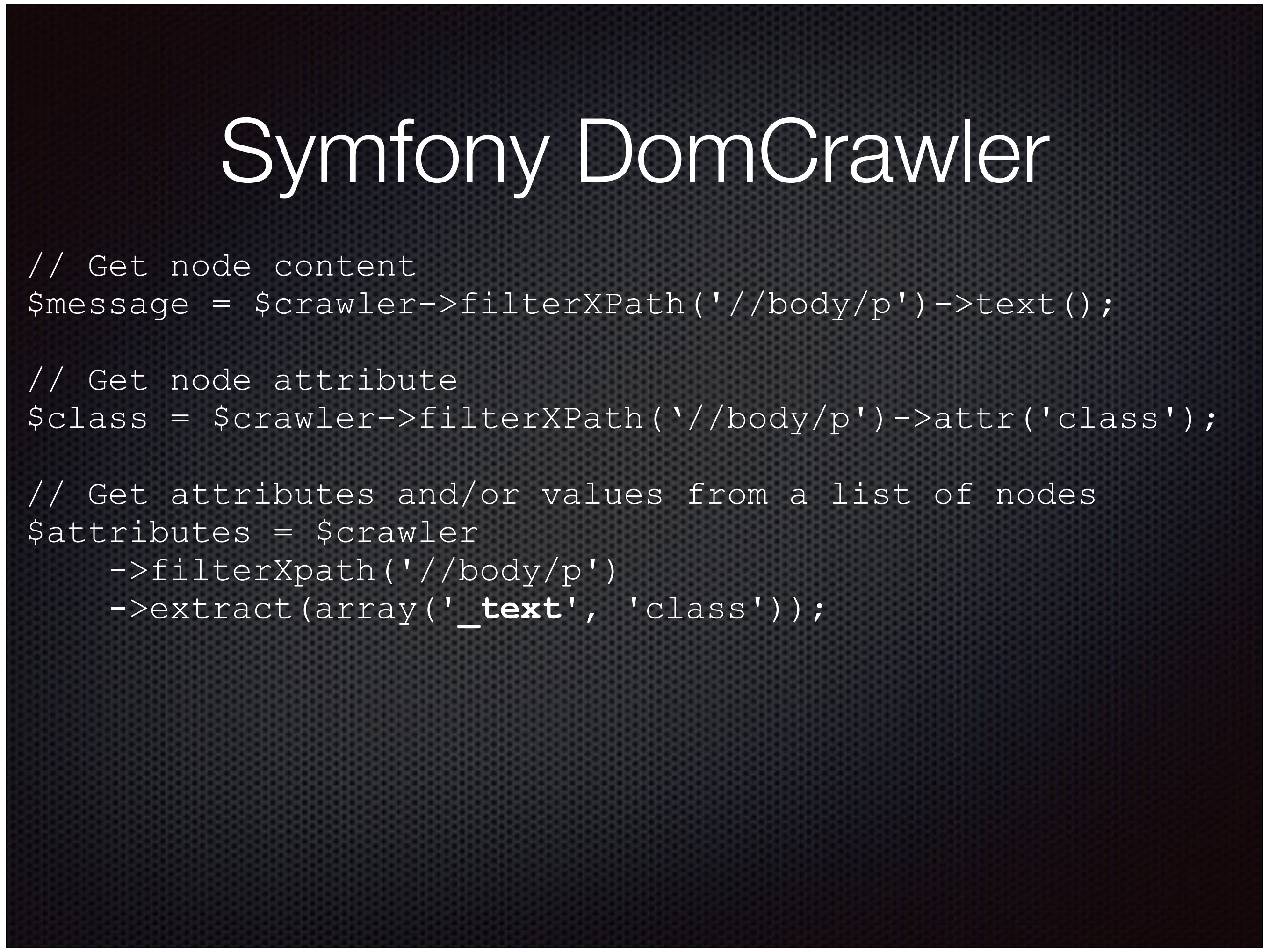
By position

```
$crawler->filter('body > p')->first();
$crawler->filter('body > p')->last();
$crawler->filter('body > p')->eq(5);
```

By relationship

```
$crawler->filter('body > p')->siblings();
$crawler->filter('body')->children();
$crawler->filter('body > p')->parents();
$crawler->filter('body > p')->nextAll();
$crawler->filter('body > p')->previousAll();
```

Once you've got your crawler, you can get nodes by position or relationship. If you've got the CssSelector component installed, which Drupal does, you can use jQuery-style selectors.

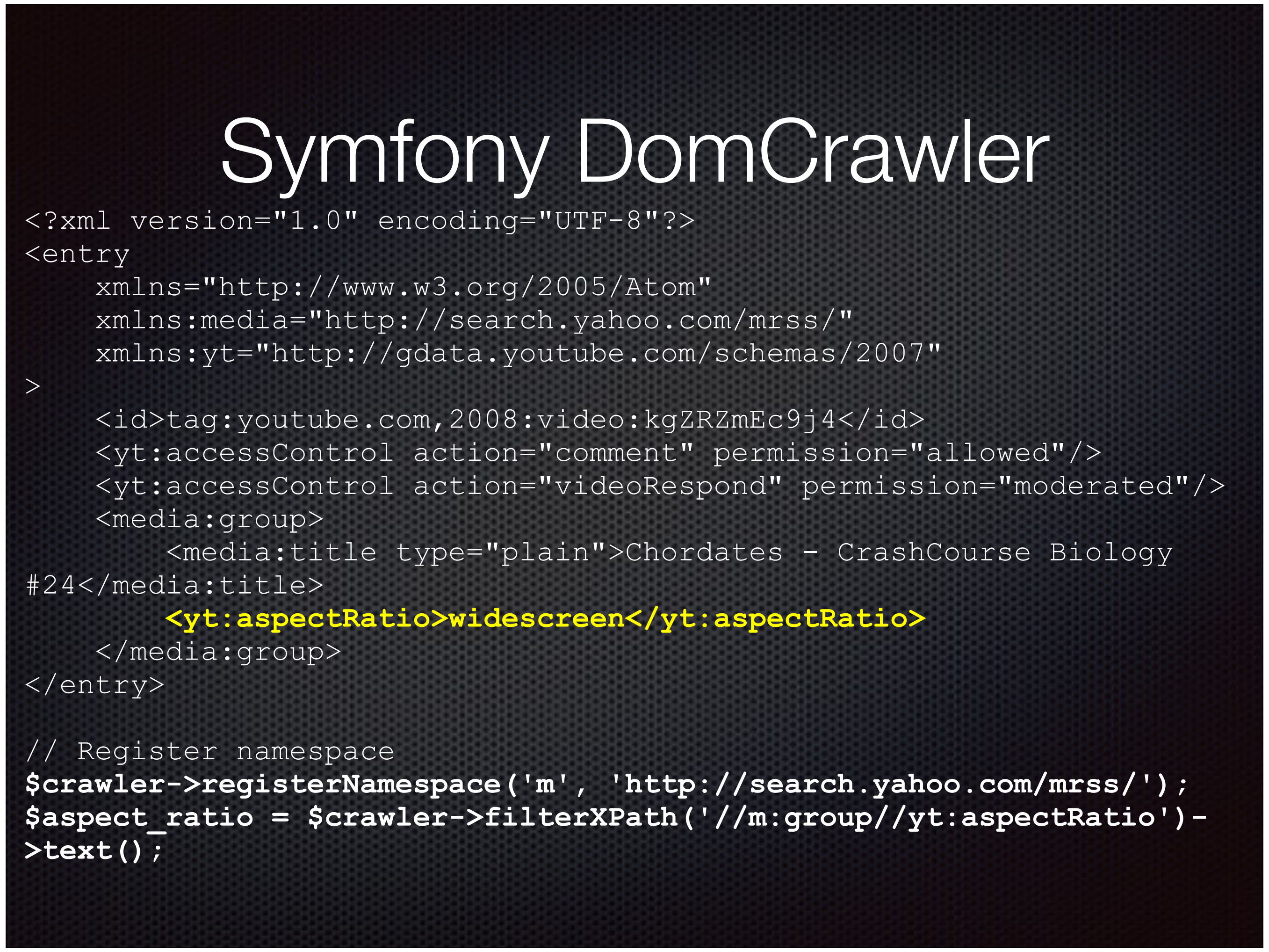


Symfony DomCrawler

```
// Get node content
$message = $crawler->filterXPath('//body/p')->text();

// Get node attribute
$class = $crawler->filterXPath(‘//body/p')->attr('class');

// Get attributes and/or values from a list of nodes
$attributes = $crawler
     ->filterXpath('//body/p')
     ->extract(array('_text', 'class'));
```

Get node content and/or attributes. Note that DomCrawler also give you an XPath filter method.

_text in the third example is kind of interesting, it’s a special attribute representing the node value



Symfony DomCrawler

```
<?xml version="1.0" encoding="UTF-8"?>
<entry
    xmlns="http://www.w3.org/2005/Atom"
    xmlns:media="http://search.yahoo.com/mrss/"
    xmlns:yt="http://gdata.youtube.com/schemas/2007"
>
    <id>tag:youtube.com,2008:video:kgZRZmEc9j4</id>
    <yt:accessControl action="comment" permission="allowed"/>
    <yt:accessControl action="videoRespond" permission="moderated"/>
    <media:group>
        <media:title type="plain">Chordates - CrashCourse Biology
#24</media:title>
        <yt:aspectRatio>widescreen</yt:aspectRatio>
    </media:group>
</entry>

// Register namespace
$crawler->registerNamespace('m', 'http://search.yahoo.com/mrss/');
$aspect_ratio = $crawler->filterXPath('//m:group//yt:aspectRatio')-
>text();
```

In this example, we want to get the aspect ratio node in an XML doc

You can register the doc’s namespace and use Xpath syntax queries

Symfony DomCrawler

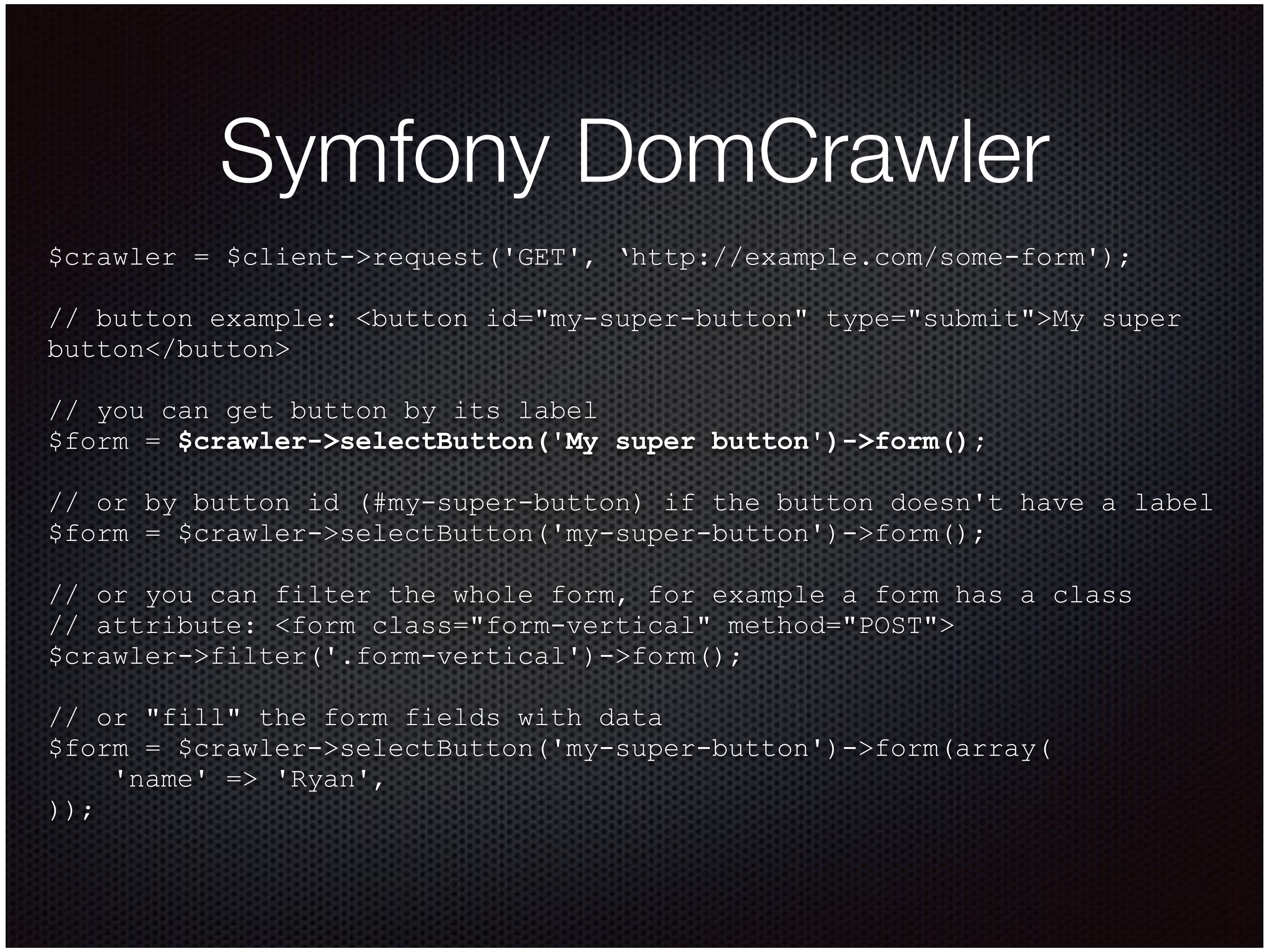
```
<?xml version="1.0" encoding="UTF-8"?>
<entry
    xmlns="http://www.w3.org/2005/Atom"
    xmlns:media="http://search.yahoo.com/mrss/"
    xmlns:yt="http://gdata.youtube.com/schemas/2007"
>
    <id>tag:youtube.com,2008:video:kgZRZmEc9j4</id>
    <yt:accessControl action="comment" permission="allowed"/>
    <yt:accessControl action="videoRespond" permission="moderated"/>
    <media:group>
        <media:title type="plain">Chordates - CrashCourse Biology
#24</media:title>
        <yt:aspectRatio>widescreen</yt:aspectRatio>
    </media:group>
</entry>

//without registering namespace
$ratio = $crawler->filterXPath('//default:entry/media:group//
yt:aspectRatio')->text();
$ratio = $crawler->filter('default|entry media|group yt|aspectRatio')-
>text();
```

Same thing without registering the namespace. Note that in this case you can use either filterXPath() or just plain filter() to get your result.

Symfony DomCrawler

```
$crawler = new Crawler('<html><body /></html>');

$crawler->addHtmlContent('<html><body /></html>');
$crawler->addXmlContent('<root><node /></root>');

$crawler->addContent('<html><body /></html>');
$crawler->addContent('<root><node /></root>', 'text/xml');

$crawler->add('<html><body /></html>');
$crawler->add('<root><node /></root>');
```

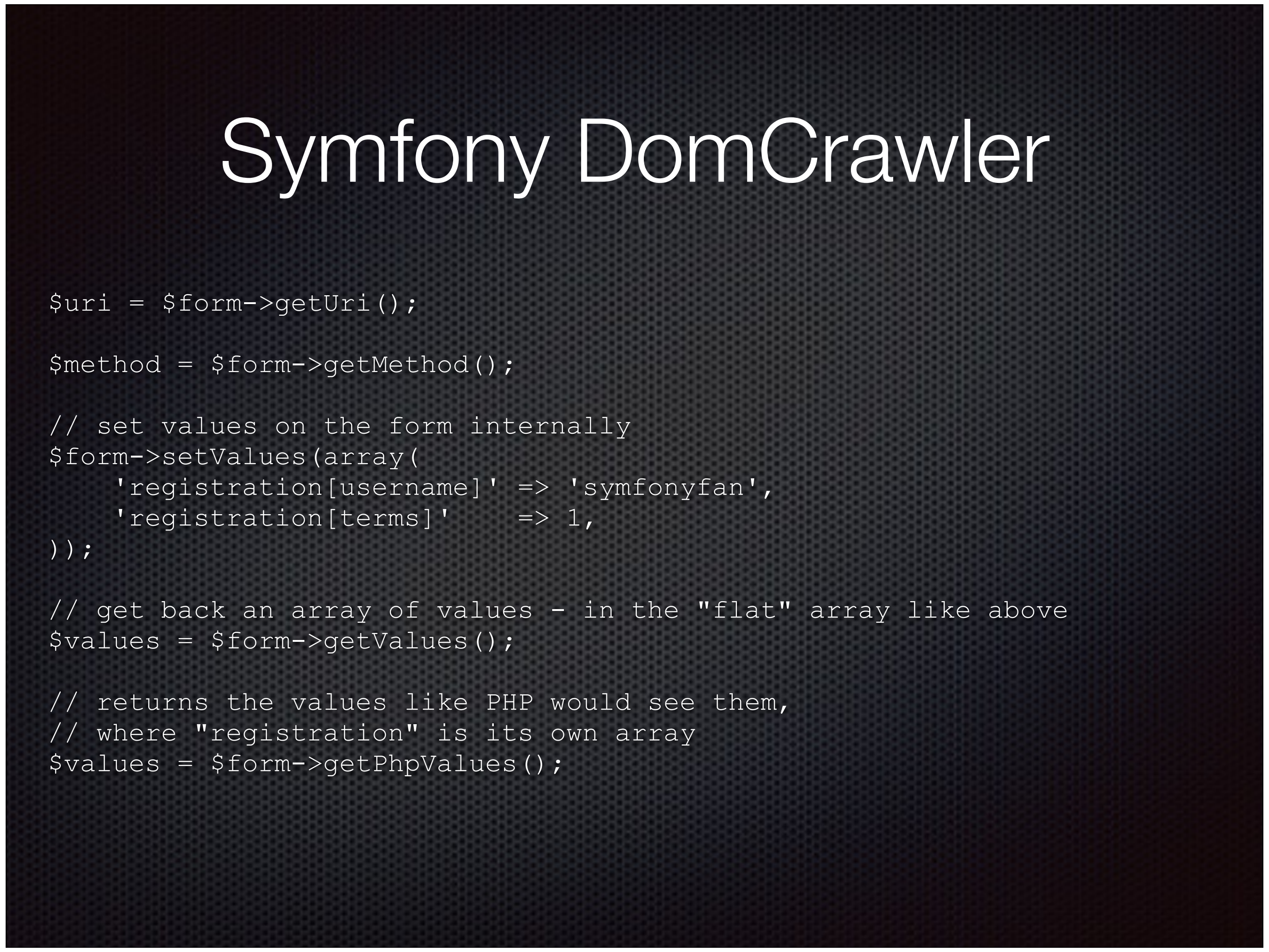
A few different methods to add content.



Symfony DomCrawler

```
$crawler = $client->request('GET', ‘http://example.com/some-form');

// button example: <button id="my-super-button" type="submit">My super
button</button>

// you can get button by its label
$form = $crawler->selectButton('My super button')->form();

// or by button id (#my-super-button) if the button doesn't have a label
$form = $crawler->selectButton('my-super-button')->form();

// or you can filter the whole form, for example a form has a class
// attribute: <form class="form-vertical" method="POST">
$crawler->filter('.form-vertical')->form();

// or "fill" the form fields with data
$form = $crawler->selectButton('my-super-button')->form(array(
    'name' => 'Ryan',
));
```

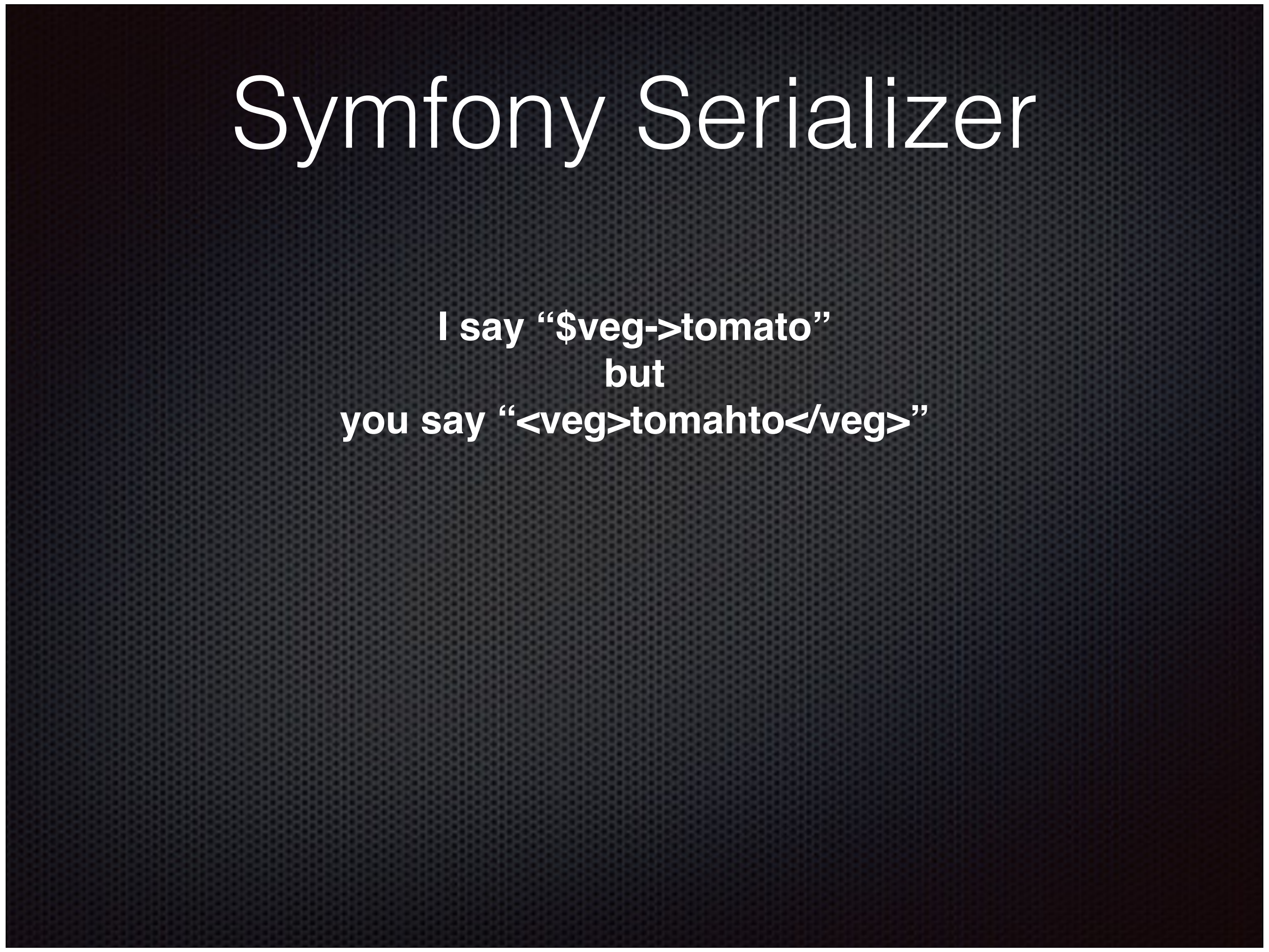
The selectButton() method is really useful if you’re crawling a form. It returns a form object that represents the form containing the button…


Symfony DomCrawler

```
$uri = $form->getUri();

$method = $form->getMethod();

// set values on the form internally
$form->setValues(array(
    'registration[username]' => 'symfonyfan',
    'registration[terms]'    => 1,
));

// get back an array of values - in the "flat" array like above
$values = $form->getValues();

// returns the values like PHP would see them,
// where "registration" is its own array
$values = $form->getPhpValues();
```

…which you can then use to get more information about the form or to get and set values.

The getUri method is pretty interesting. If the form method is a GET, getUri returns the action and a query string of all the form's values.



Symfony Serializer

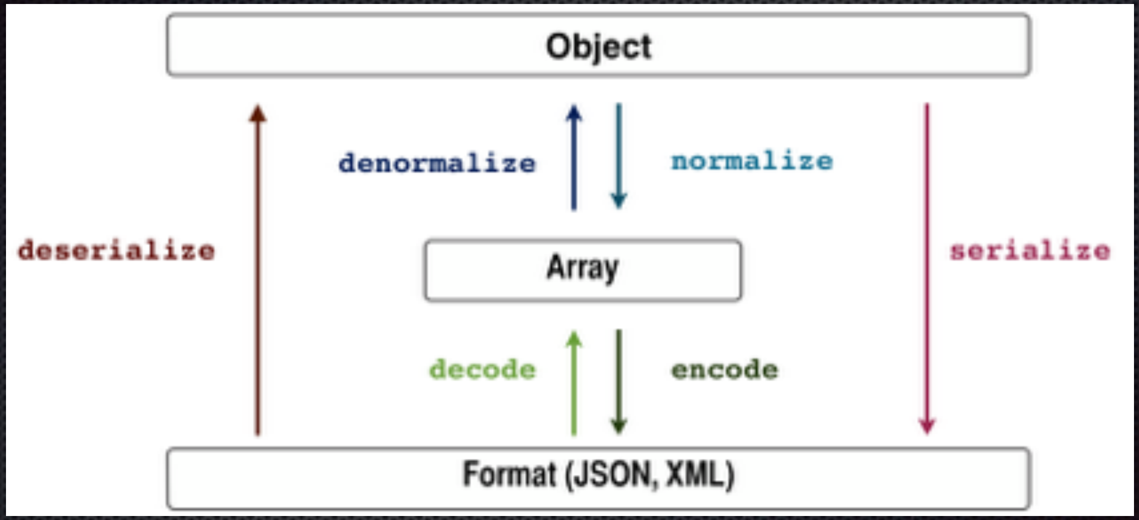
I say “$veg->tomato”
but
you say “<veg>tomahto</veg>”

Converting data from one format to another is one of the most common situations in modern web development. It’s definitely something we do a lot of in the group I work with.

You ingest some XML and need to do something with it, or, especially now with headless implementations, you need to get some data from Drupal and deliver it via an API to a JS front end.

For that, there’s the Serializer component.

Symfony Serializer

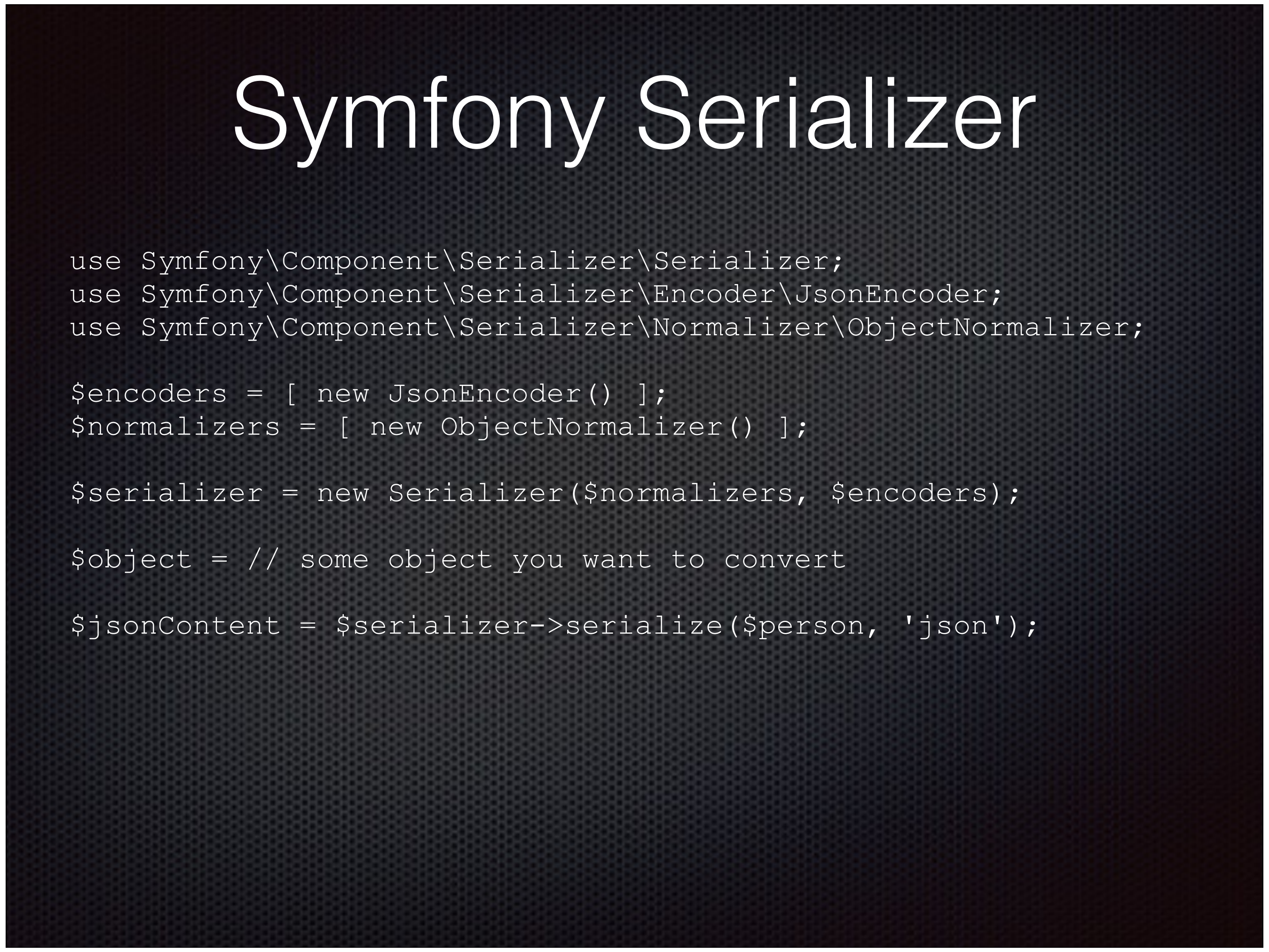


Symfony serializer lets you convert an object to a format, and vice versa.

It involves a couple of steps.

A normalizer turns an object into an array, while an encoder converts an array into a format.

Going the other way, a decoder turns a format into an array a denormalizer turns that array into an object.



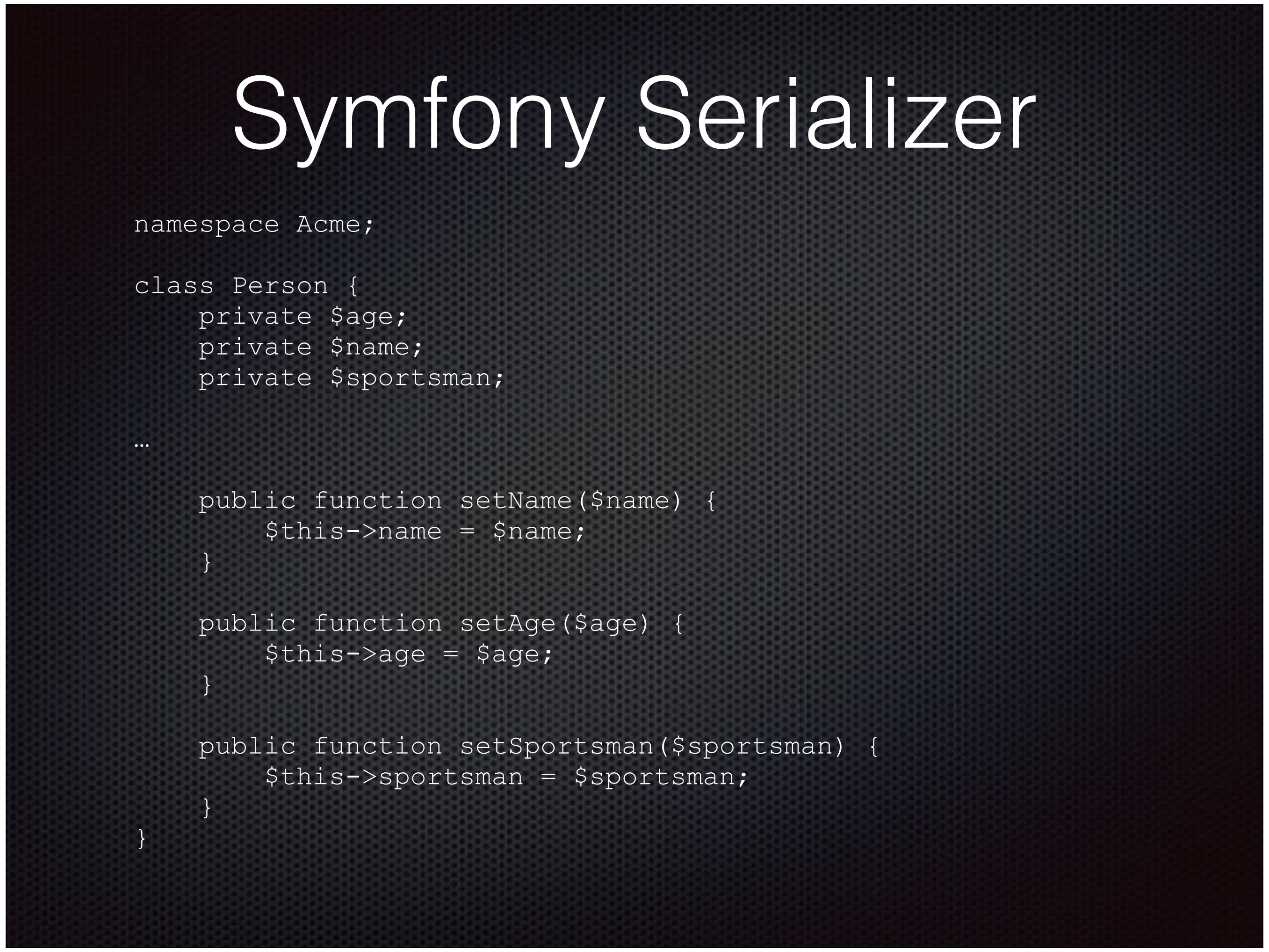
Symfony Serializer

```
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;

$encoders = [ new JsonEncoder() ];
$normalizers = [ new ObjectNormalizer() ];

$serializer = new Serializer($normalizers, $encoders);

$object = // some object you want to convert

$jsonContent = $serializer->serialize($person, 'json');
```

To use the serializer component, you set up the serializer specifying which encoders and normalizer you want to use. Here we're using Symfony's JsonEncoder and ObjectNormalizer.

Once you've got your serializer instantiated and have an object you want to convert, pass the object to the serializer's serialize() method, along with the format you want to turn it into.

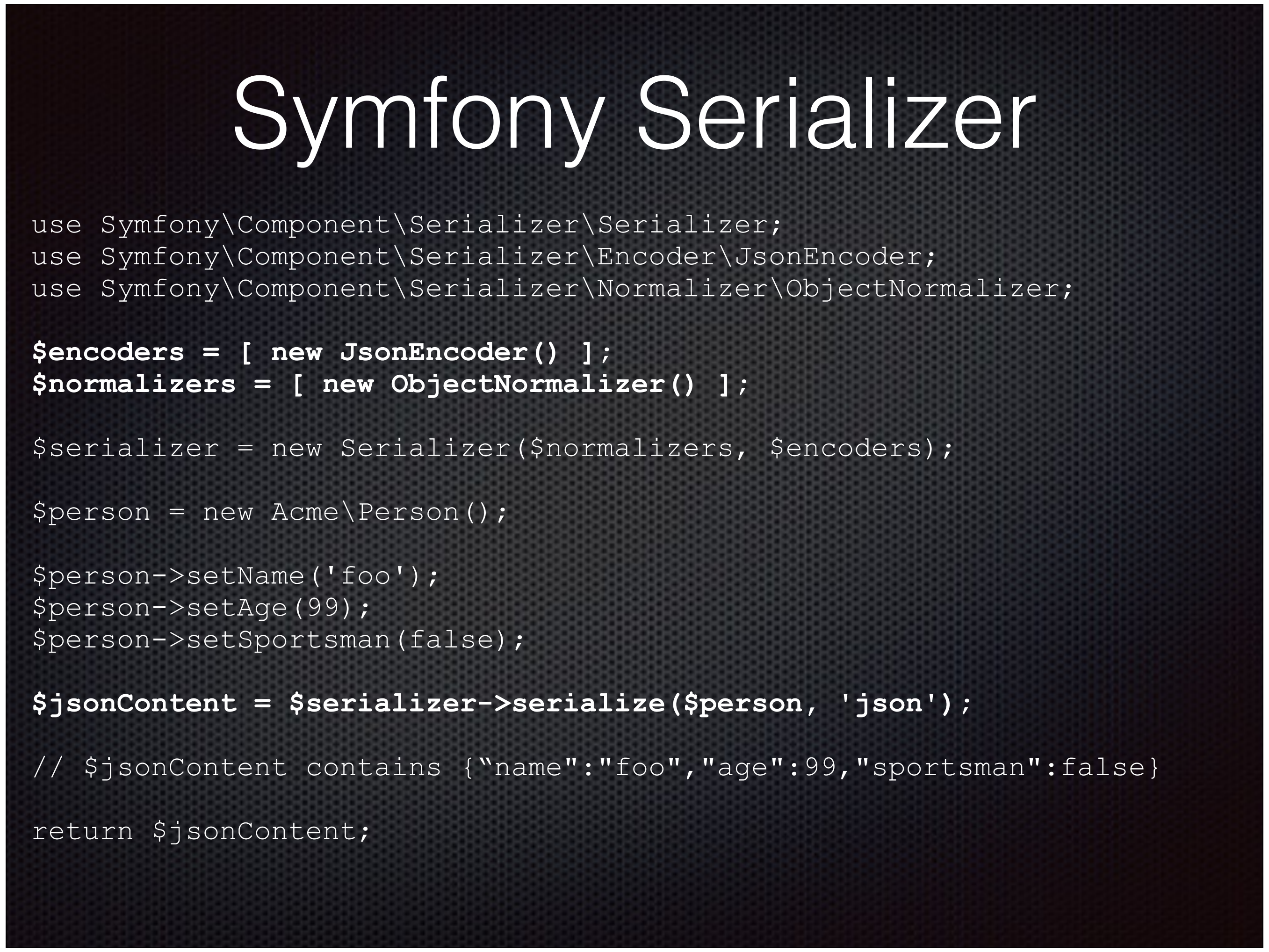
Let's look at an example.



Symfony Serializer

```
namespace Acme;

class Person {
    private $age;
    private $name;
    private $sportsman;

…

    public function setName($name) {
        $this->name = $name;
    }

    public function setAge($age) {
        $this->age = $age;
    }

    public function setSportsman($sportsman) {
        $this->sportsman = $sportsman;
    }
}
```

Let's say we have this class. We're headless, so we need an API to be able to send Persons to our React front end.



Symfony Serializer

```
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;

$encoders = [ new JsonEncoder() ];
$normalizers = [ new ObjectNormalizer() ];

$serializer = new Serializer($normalizers, $encoders);

$person = new Acme\Person();

$person->setName('foo');
$person->setAge(99);
$person->setSportsman(false);

$jsonContent = $serializer->serialize($person, 'json');

// $jsonContent contains {“name":"foo","age":99,"sportsman":false}

return $jsonContent;
```

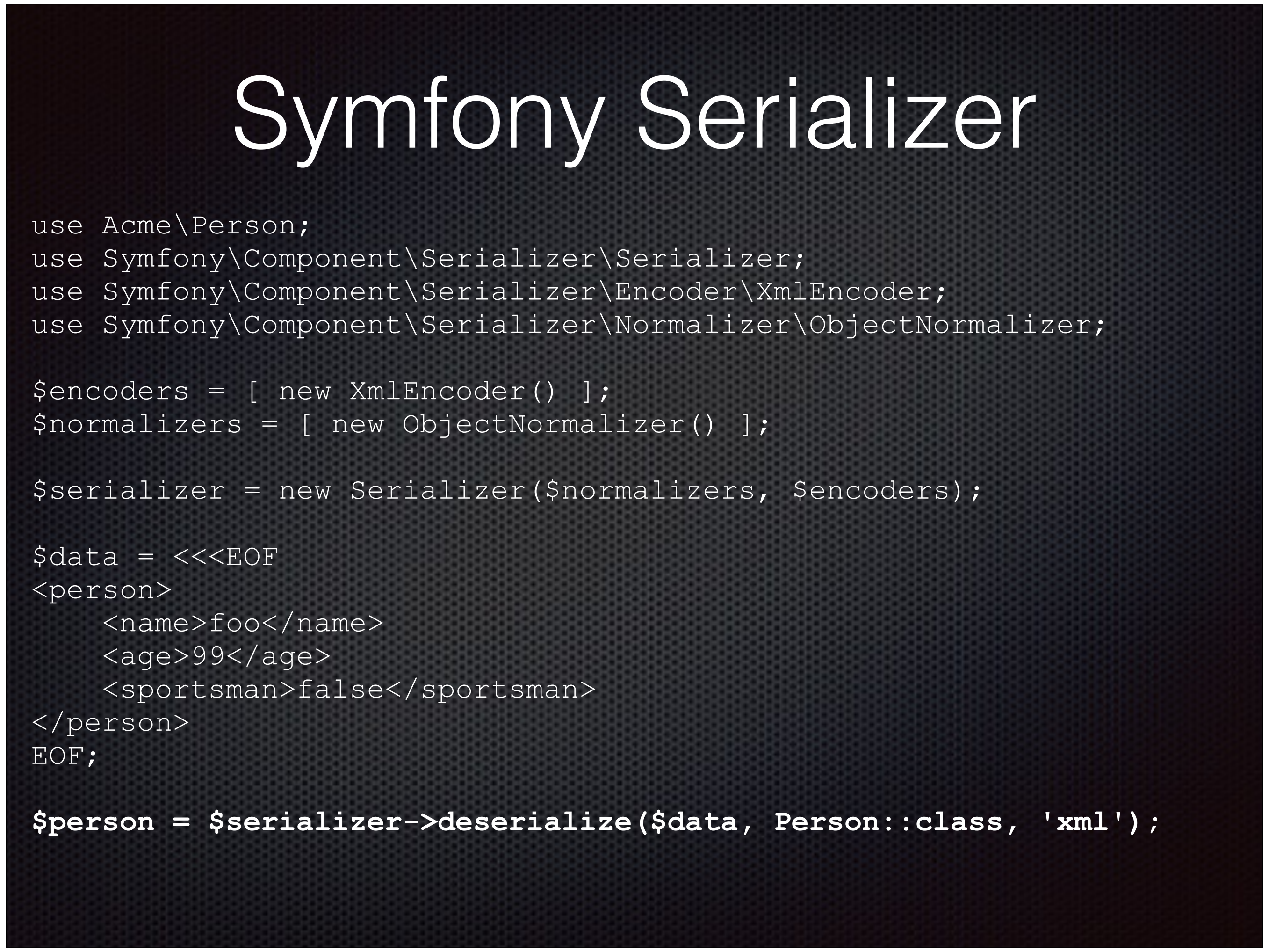
So, our Api class declares the encoders and normalizers we need and set up a Serializer object. In this case, we want JSON, so we use JsonEncoder. We then tell the serializer to encode the $person object using the json encoder and return that.

You might recall that HttpFoundation has a JsonResponse that would probably be fine for a simple example like this, but real life isn’t always this simple.

Symfony Serializer

```
use Symfony\Component\Serializer\Encoder\JsonEncode;

$person = new Acme\Person();

$person->setName('foo');
$person->setAge(99);
$person->setSportsman(false);

$encoder = new JsonEncode();

$jsonContent = $encoder->encode($person);

// $jsonContent contains {“name”:"foo","age":99,"sportsman":false}

return $jsonContent;
```

In keeping with the tradition of showing the hard way before showing the easy way—
For JSON, you can also just use JsonEncode. Symfony also provides a JsonDecode class that you’d use the same way.

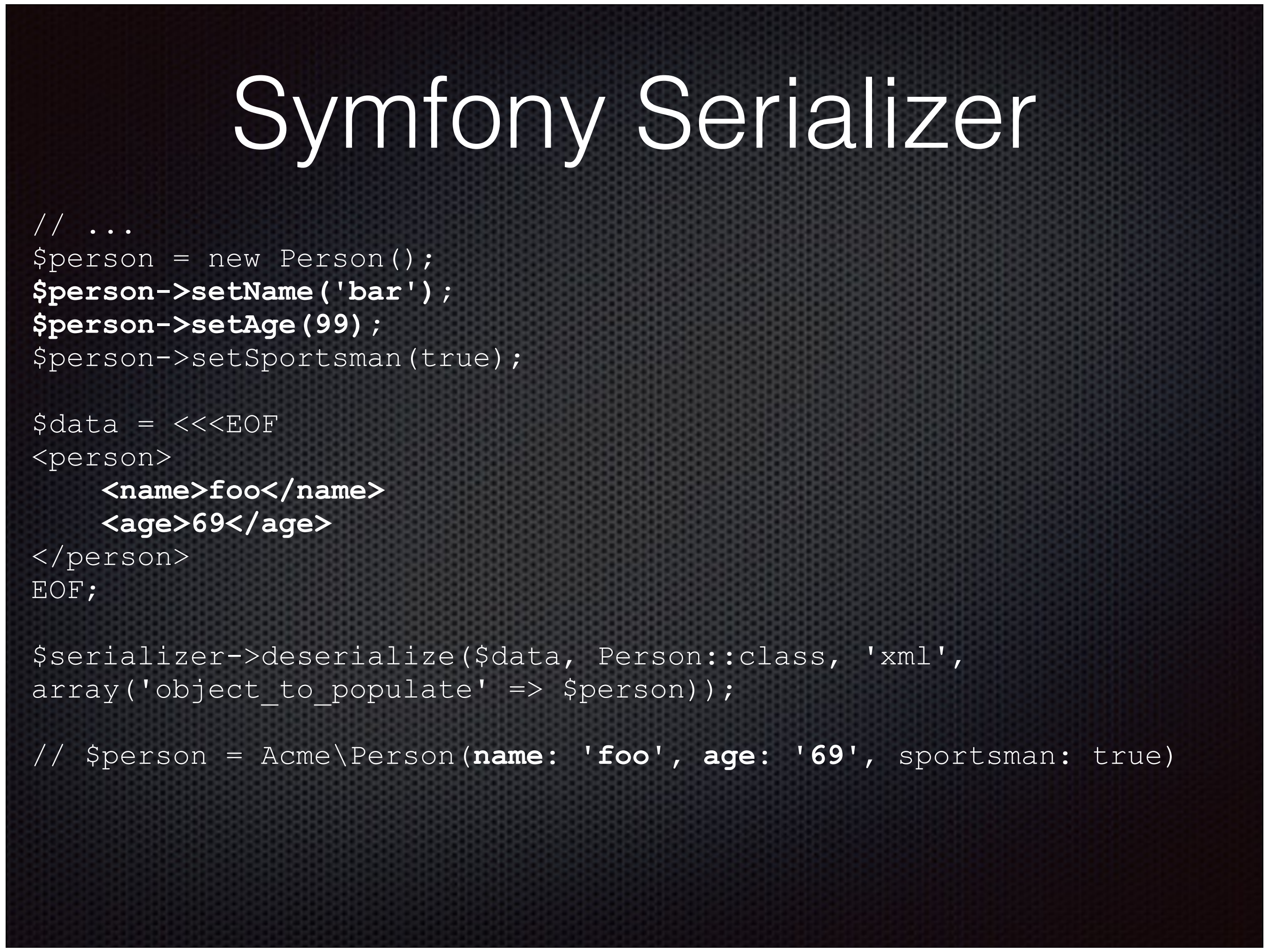


Symfony Serializer

```
use Acme\Person;
use Symfony\Component\Serializer\Serializer;
use Symfony\Component\Serializer\Encoder\XmlEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;

$encoders = [ new XmlEncoder() ];
$normalizers = [ new ObjectNormalizer() ];

$serializer = new Serializer($normalizers, $encoders);

$data = <<<EOF
<person>
    <name>foo</name>
    <age>99</age>
    <sportsman>false</sportsman>
</person>
EOF;

$person = $serializer->deserialize($data, Person::class, 'xml');
```

Now let’s go the other way.

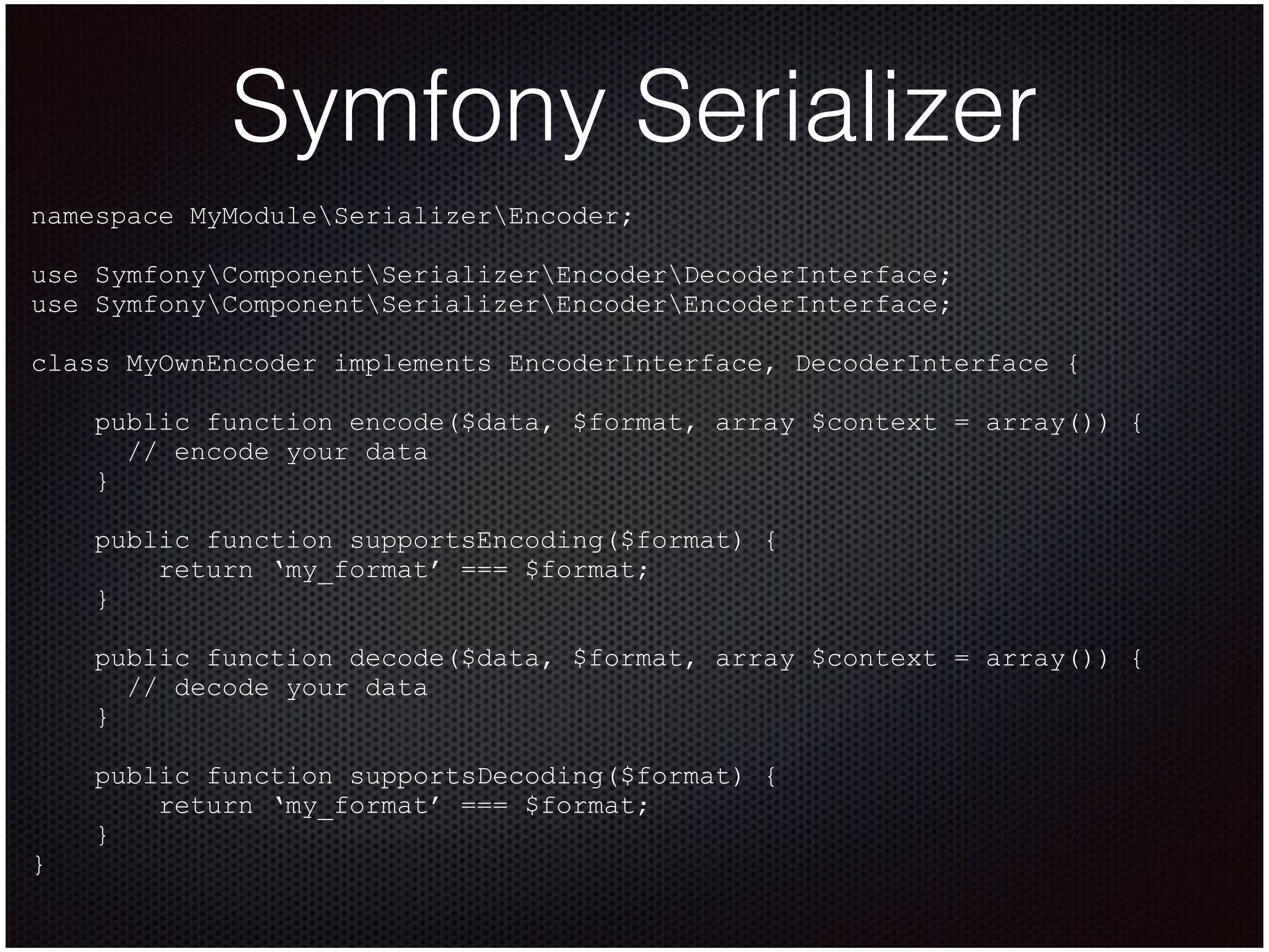
You’re calling a REST service that gives you Persons in XML format and you need to create Person objects to store in your Drupal DB. In this case, you call the encoder’s deserialize() method, giving it the incoming data, the class you want to deserialize to and the encoder to use.



Symfony Serializer

```
// ...
$person = new Person();
$person->setName('bar');
$person->setAge(99);
$person->setSportsman(true);

$data = <<<EOF
<person>
    <name>foo</name>
    <age>69</age>
</person>
EOF;

$serializer->deserialize($data, Person::class, 'xml',
array('object_to_populate' => $person));

// $person = Acme\Person(name: 'foo', age: '69', sportsman: true)
```

Serializer can also update an existing object by accepting an additional parameter to deserialize(). In this example, incoming data changes the Person’s name and age.



Symfony Serializer

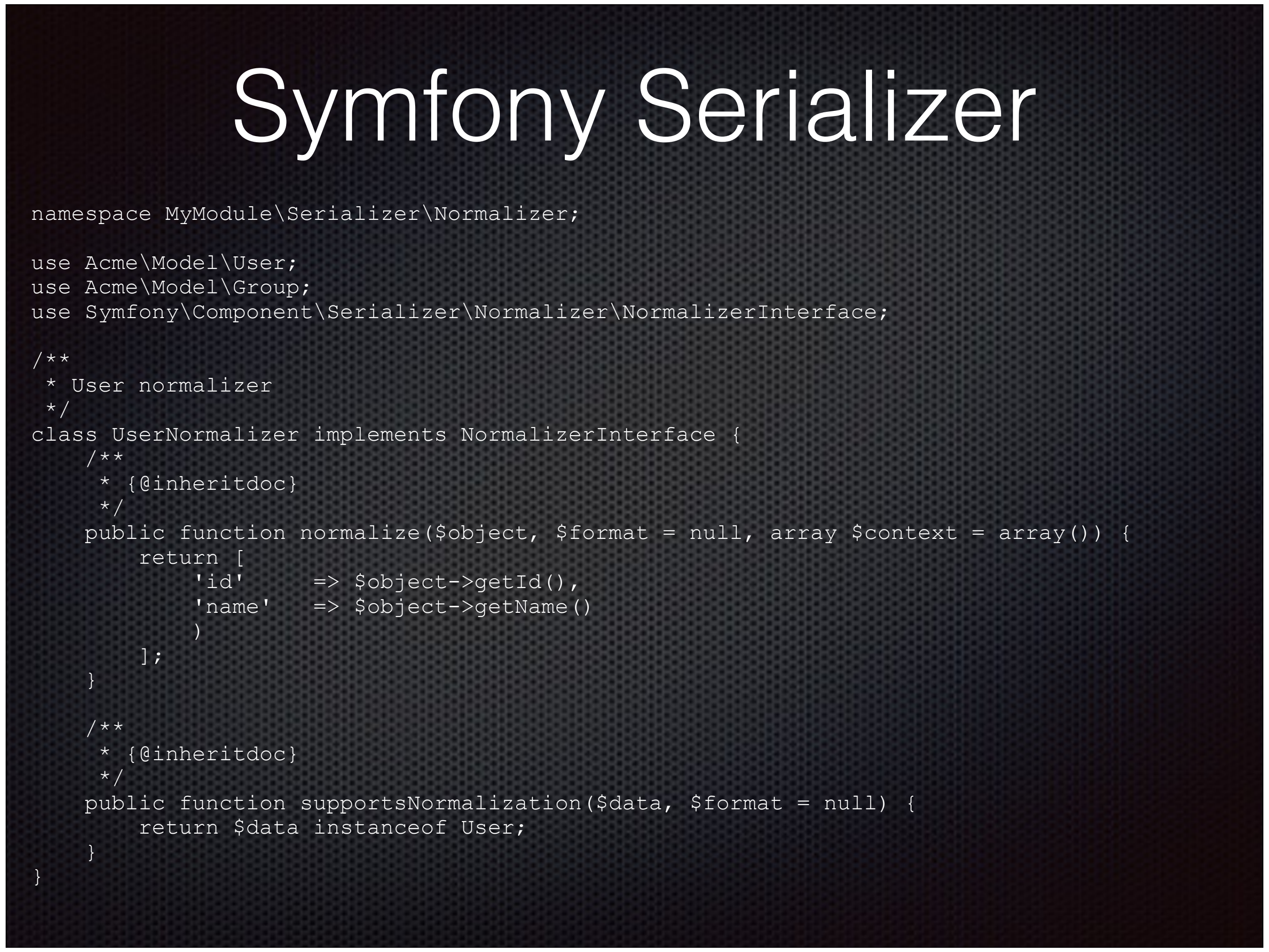
```
namespace MyModule\Serializer\Encoder;

use Symfony\Component\Serializer\Encoder\DecoderInterface;
use Symfony\Component\Serializer\Encoder\EncoderInterface;

class MyOwnEncoder implements EncoderInterface, DecoderInterface {

    public function encode($data, $format, array $context = array()) {
      // encode your data
    }

    public function supportsEncoding($format) {
        return 'my_format' === $format;
    }

    public function decode($data, $format, array $context = array()) {
      // decode your data
    }

    public function supportsDecoding($format) {
        return 'my_format' === $format;
    }
}
```

Symfony has built-in encoders for JSON and XML, and Symfony 3.2 adds decoders for Yaml and CSV.

You can create your own encoder/decoder for other formats by implementing the Encoder and Decoder Interfaces.

The EncoderInterface implementation tells what format your encoder supports and provides an encode() method.

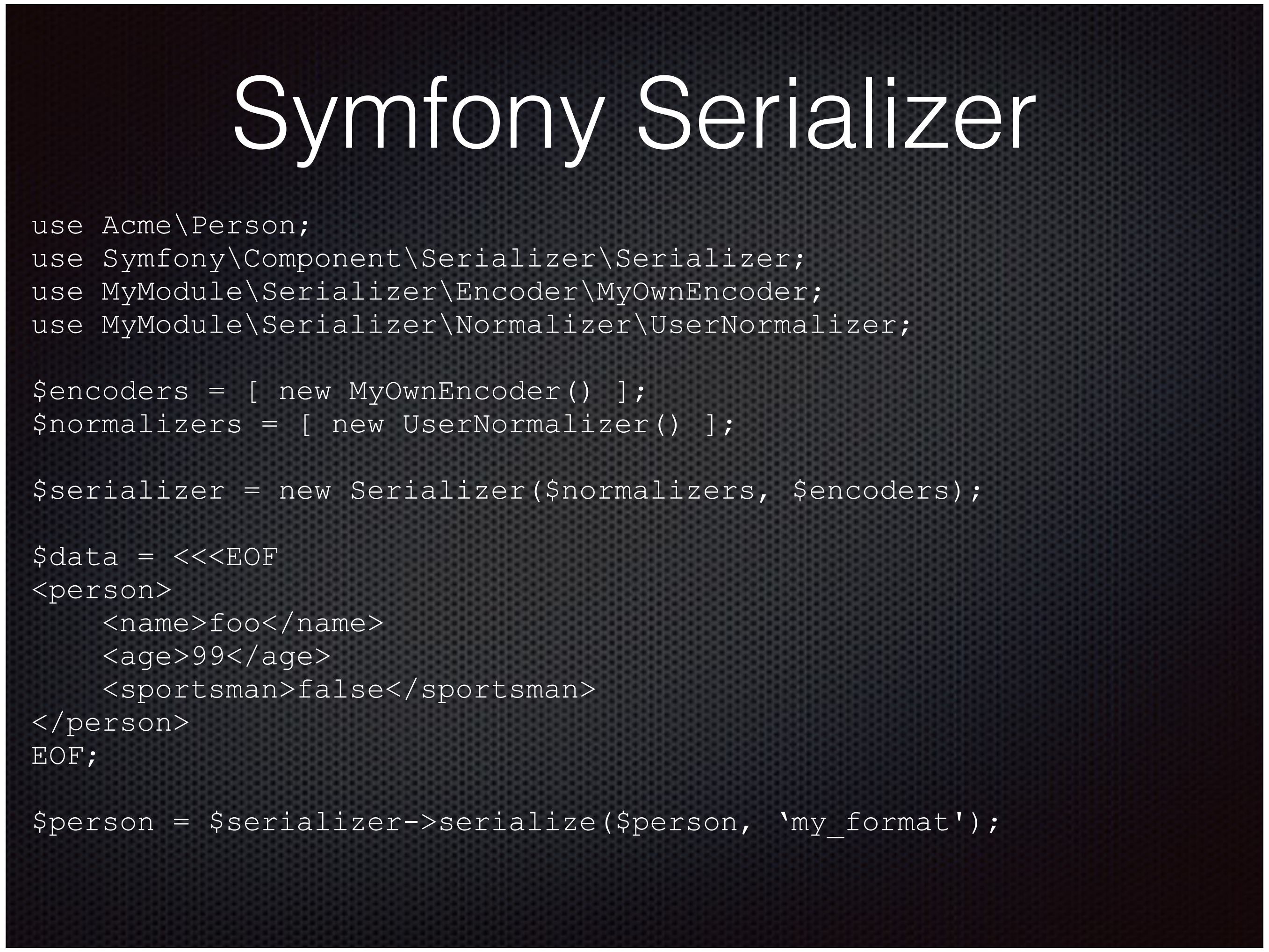
The DecoderInterface implementation tells what format you can decode and provides a decode() method.



Symfony Serializer

```
namespace MyModule\Serializer\Normalizer;

use Acme\Model\User;
use Acme\Model\Group;
use Symfony\Component\Serializer\Normalizer\NormalizerInterface;

/**
 * User normalizer
 */
class UserNormalizer implements NormalizerInterface {
    /**
     * {@inheritdoc}
     */
    public function normalize($object, $format = null, array $context = array()) {
        return [
            'id'     => $object->getId(),
            'name'   => $object->getName()
            )
        ];
    }

    /**
     * {@inheritdoc}
     */
    public function supportsNormalization($data, $format = null) {
        return $data instanceof User;
    }
}
```

Likewise, you can create your own normalizer, implementing NormalizerInterface, that takes an object and maps it to an array. The structure here will determine the structure of your serialized data. For instance if you're serializing to XML, array keys correspond to XML nodes.



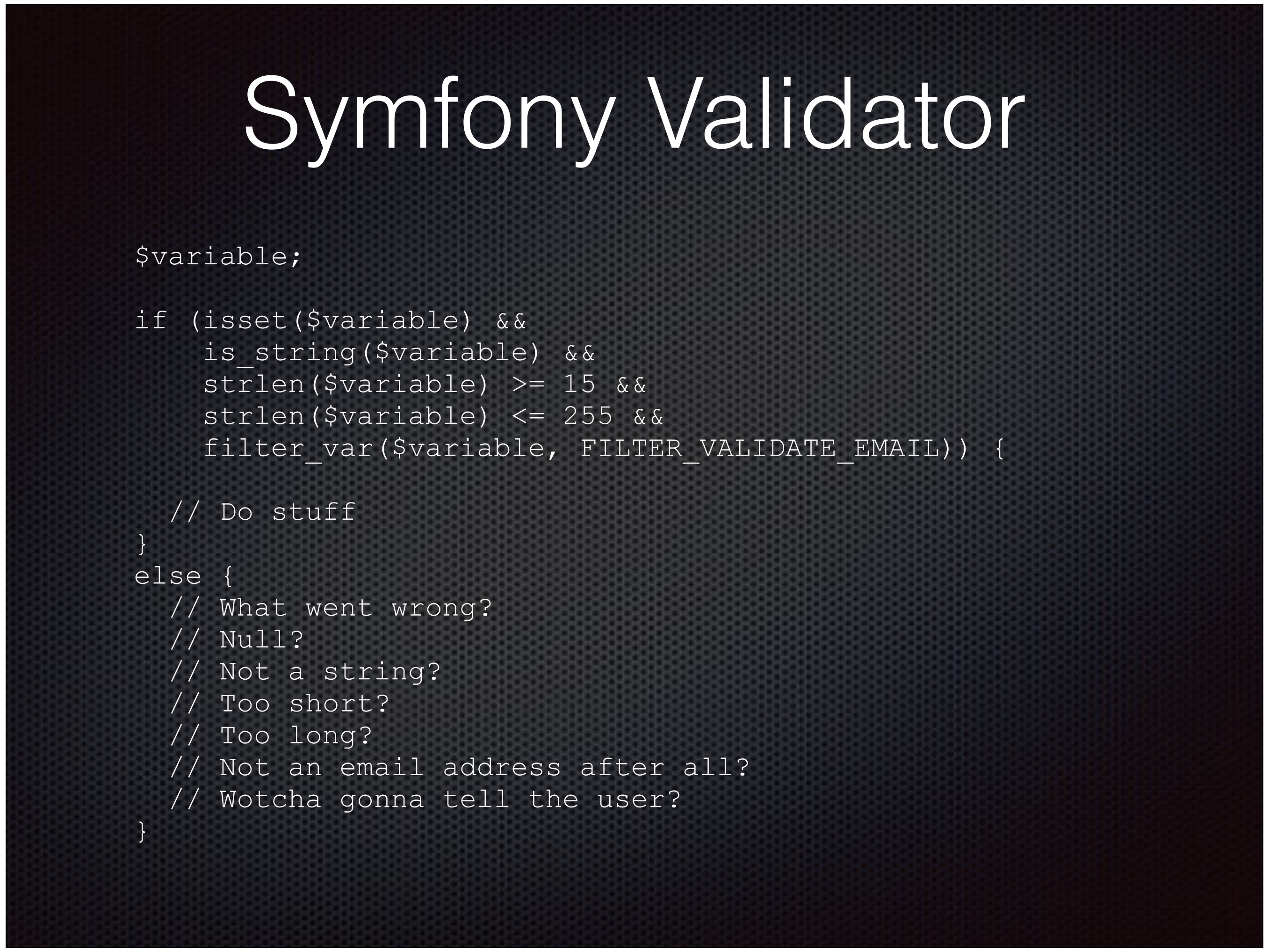
Symfony Serializer

```
use Acme\Person;
use Symfony\Component\Serializer\Serializer;
use MyModule\Serializer\Encoder\MyOwnEncoder;
use MyModule\Serializer\Normalizer\UserNormalizer;

$encoders = [ new MyOwnEncoder() ];
$normalizers = [ new UserNormalizer() ];

$serializer = new Serializer($normalizers, $encoders);

$data = <<<EOF
<person>
    <name>foo</name>
    <age>99</age>
    <sportsman>false</sportsman>
</person>
EOF;

$person = $serializer->serialize($person, ‘my_format');
```

Now you can tell Serializer to use your normalizer and encoder just like one of Symfony’s built-ins.

Drupal’s HAL module is a good one to check out for the real details on how to implement a custom format, so if you’re interested in learning more about serializing, that’s a good place to start.

Meanwhile, we’ve got one more component, and this one might be my favorite of all I’ve found out about on this little journey.



Symfony Validator

```
$variable;

if (isset($variable) &&
    is_string($variable) &&
    strlen($variable) >= 15 &&
    strlen($variable) <= 255 &&
    filter_var($variable, FILTER_VALIDATE_EMAIL)) {

  // Do stuff
}
else {
  // What went wrong?
  // Null?
  // Not a string?
  // Too short?
  // Too long?
  // Not an email address after all?
  // Wotcha gonna tell the user?
}
```

I give you the Validator component.

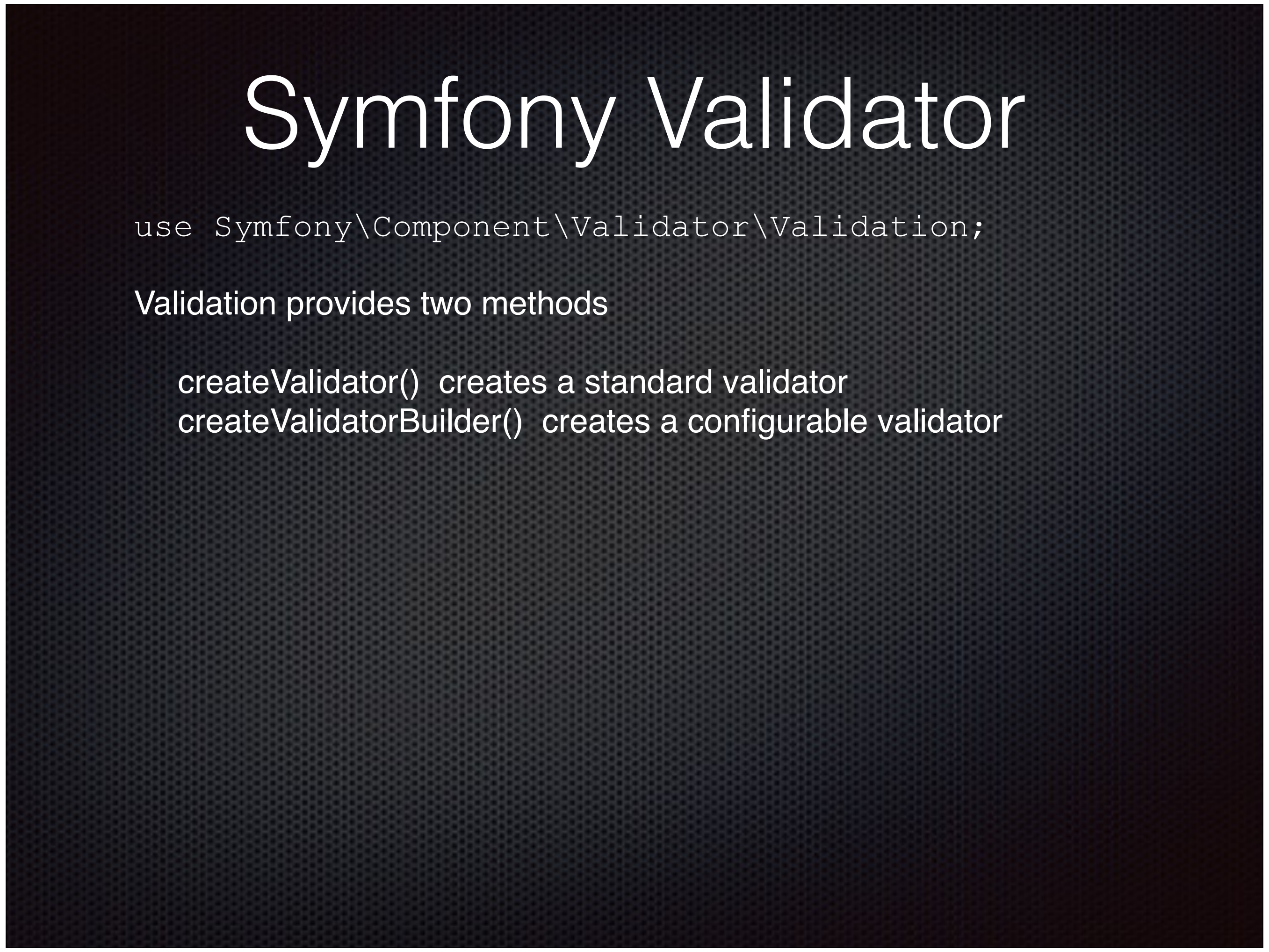
You've got this variable. You might not know where it came from, but you do know that, in order not to break your code, it can't be null, it's supposed to be a string, it's supposed to be between 15 and 255 characters long, and it's supposed to be a valid email address.

OK, not so bad to make sure all 4 conditions are true and do stuff if they are. Not bad, but certainly not pretty.

And what if one of those fails? What are you going to tell the user? They're probably going to want more detail than just that something went wrong. That's going to make your else block pretty awkward.

And what if you need to do the same thing elsewhere in your code? You can wrap all this up in a class method and use that class whenever you need to do this check. You could do that, but it's still kinda ugly.

That's why you really should be using Validator instead.



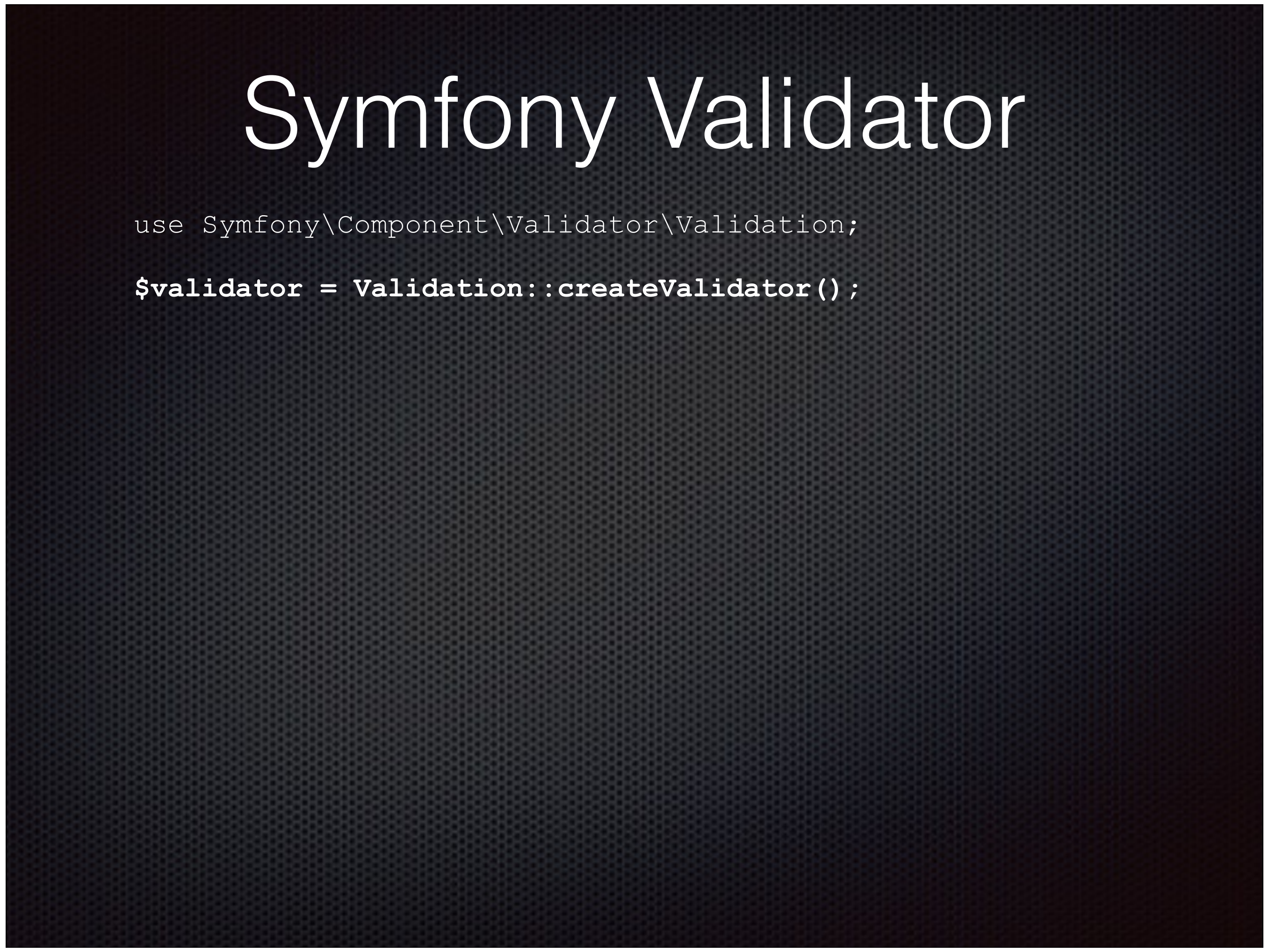
Symfony Validator

```
use Symfony\Component\Validator\Validation;
```

Validation provides two methods

createValidator() creates a standard validator
createValidatorBuilder() creates a configurable validator

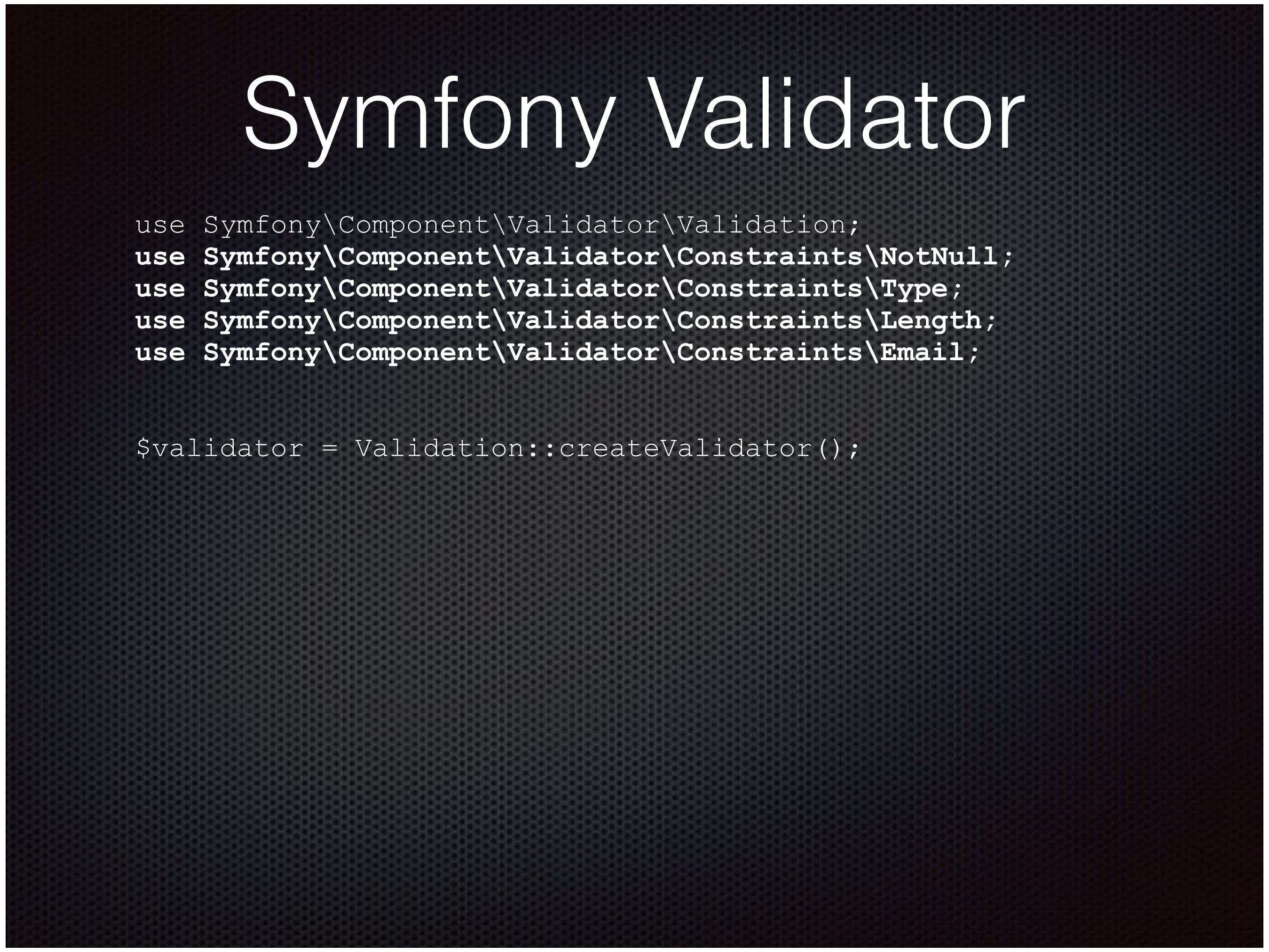
It’s easy. The Validation class is the entry point, so include that wherever you want to validate something.



Symfony Validator

```
use Symfony\Component\Validator\Validation;

$validator = Validation::createValidator();
```

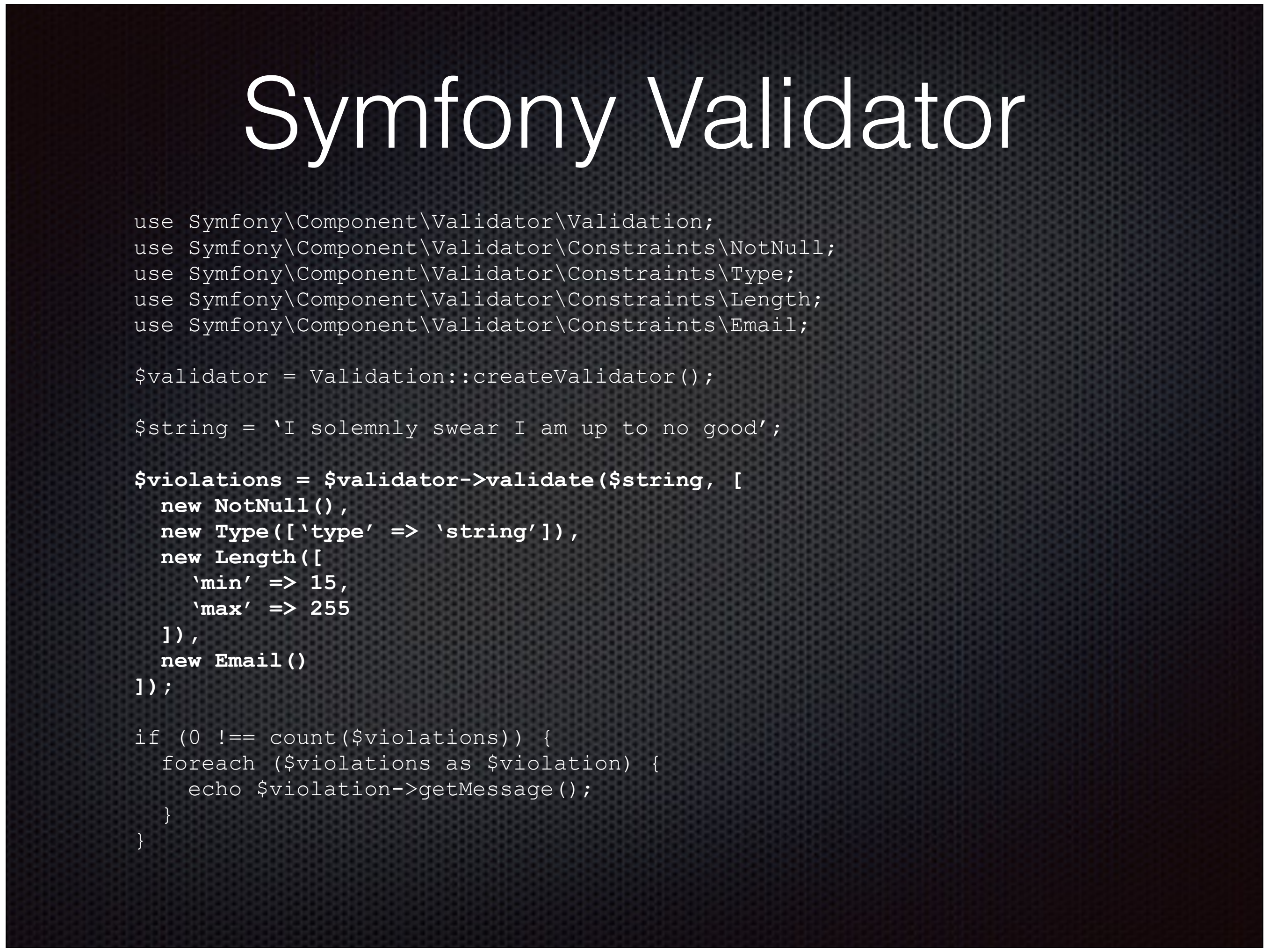
Now create a validator. We're keeping it simple here and using the plain old createValidator.



Symfony Validator

```
use Symfony\Component\Validator\Validation;
use Symfony\Component\Validator\Constraints\NotNull;
use Symfony\Component\Validator\Constraints\Type;
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\Email;


$validator = Validation::createValidator();
```

Now that you've got a validator, add some constraints. We're going to check for nullity, stringiness, length and email validity, and Symfony has constraints for all of those.



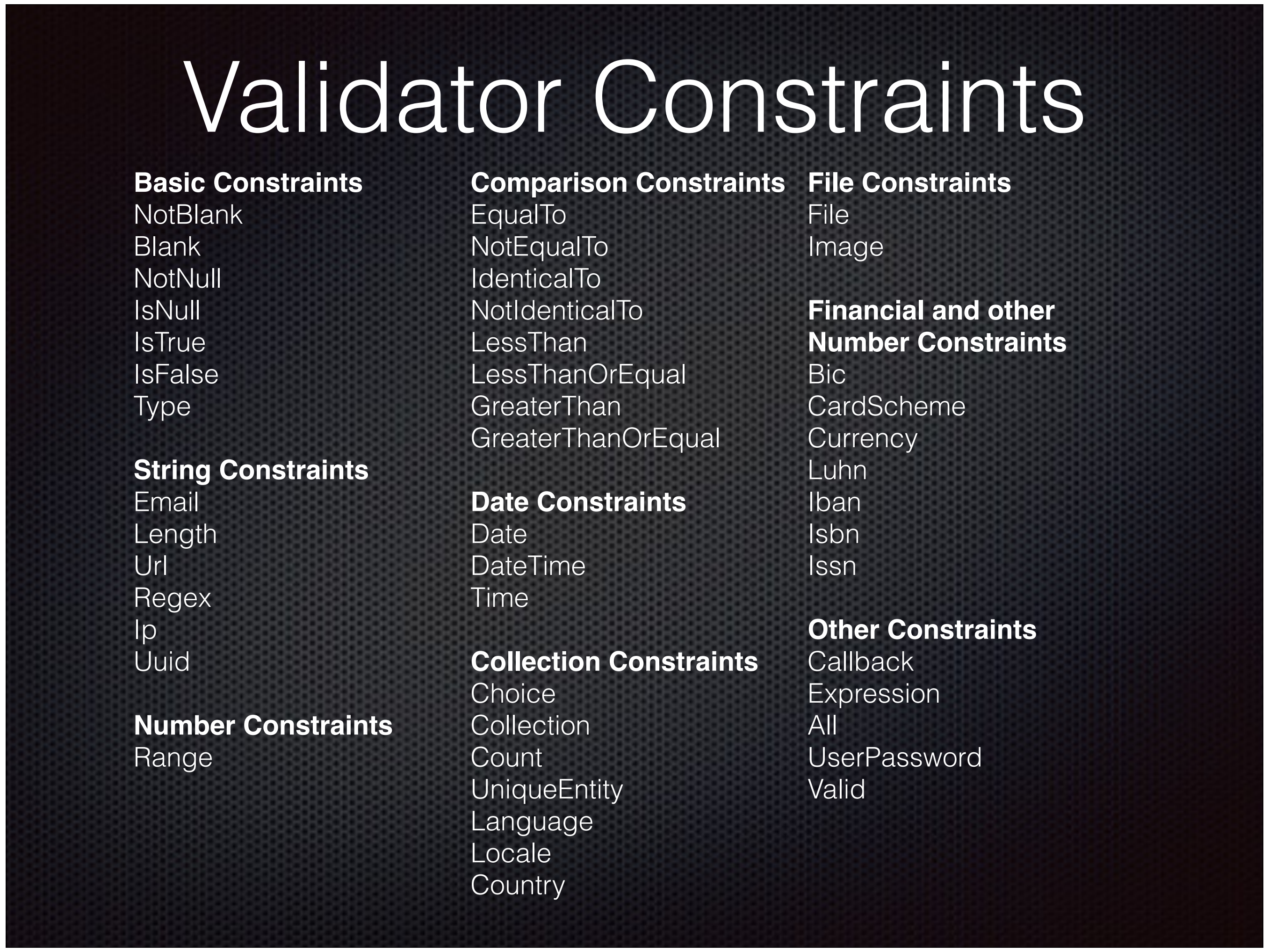
Symfony Validator

```
use Symfony\Component\Validator\Validation;
use Symfony\Component\Validator\Constraints\NotNull;
use Symfony\Component\Validator\Constraints\Type;
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\Email;

$validator = Validation::createValidator();

$string = ‘I solemnly swear I am up to no good’;

$violations = $validator->validate($string, [
  new NotNull(),
  new Type([‘type’ => ‘string’]),
  new Length([
    ‘min’ => 15,
    ‘max’ => 255
  ]),
  new Email()
]);

if (0 !== count($violations)) {
  foreach ($violations as $violation) {
    echo $violation->getMessage();
  }
}
```

And validate.

Call $validator->validate with the variable you want to validate and an array of constraint objects to validate against.

Constraints provide their own messages when a constraint fails, so all you need to know about is if there were any constraint violations.



Validator Constraints

Basic Constraints
NotBlank
Blank
NotNull
IsNull
IsTrue
IsFalse
Type

String Constraints
Email
Length
Url
Regex
Ip
Uuid

Number Constraints
Range

Comparison Constraints
EqualTo
NotEqualTo
IdenticalTo
NotIdenticalTo
LessThan
LessThanOrEqual
GreaterThan
GreaterThanOrEqual

Date Constraints
Date
DateTime
Time

Collection Constraints
Choice
Collection
Count
UniqueEntity
Language
Locale
Country

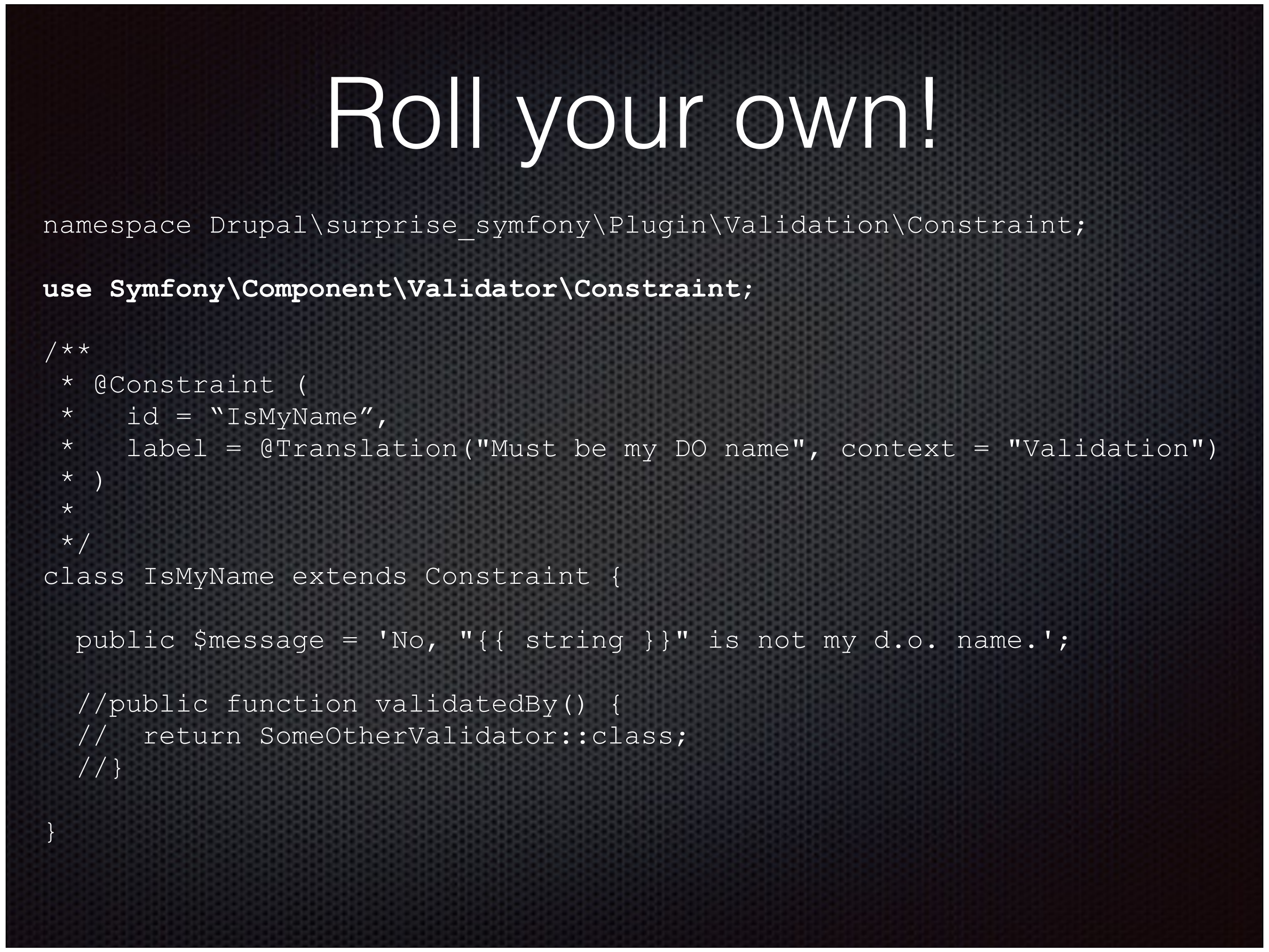
File Constraints
File
Image

Financial and other Number Constraints
Bic
CardScheme
Currency
Luhn
Iban
Isbn
Issn

Other Constraints
Callback
Expression
All
UserPassword
Valid

Symfony has lots of built-in constraints.

But if those aren’t enough, you can easily create your own.



Roll your own!

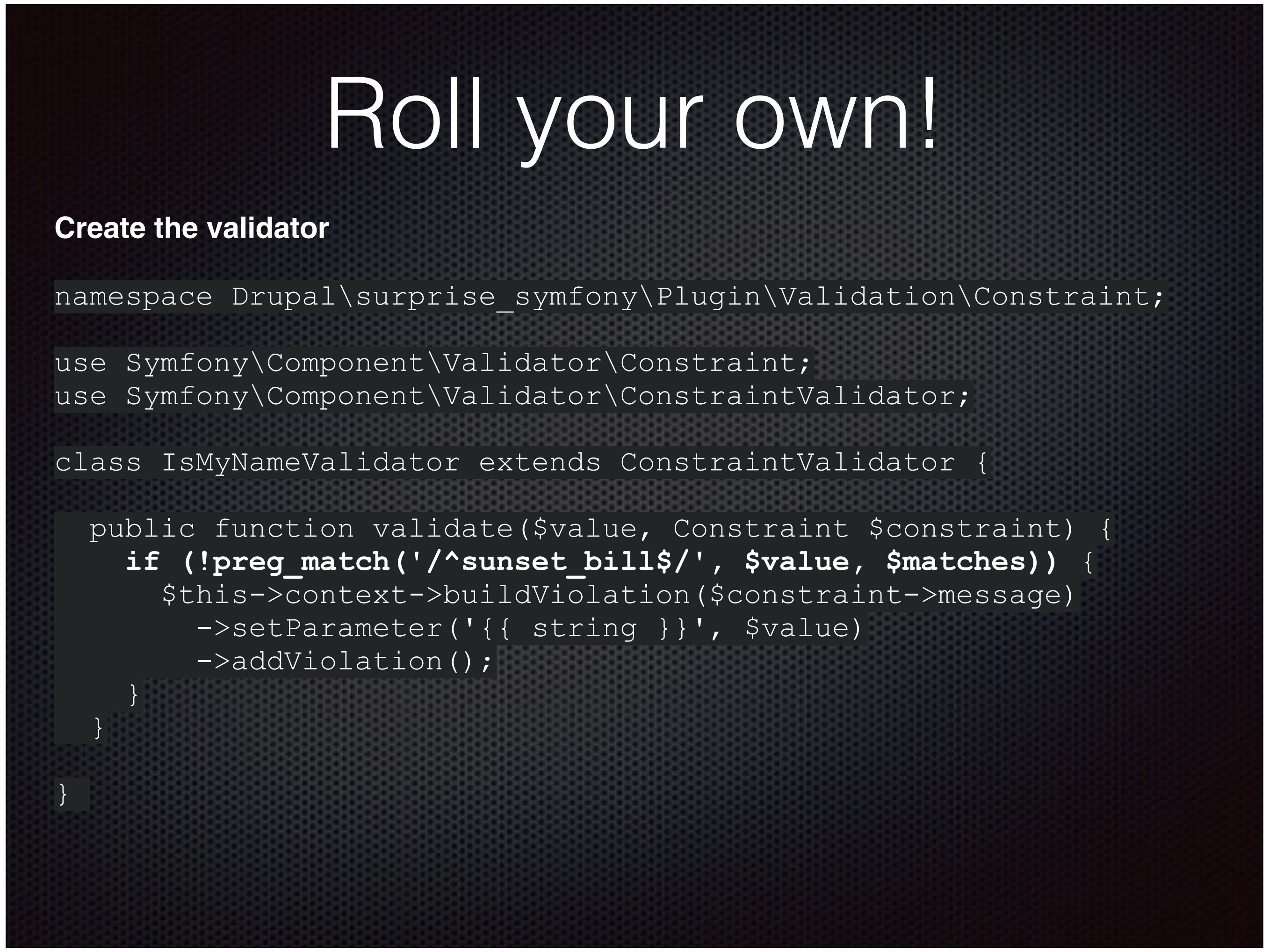
```
namespace Drupal\surprise_symfony\Plugin\Validation\Constraint;

use Symfony\Component\Validator\Constraint;

/**
 * @Constraint (
 *   id = “IsMyName”,
 *   label = @Translation("Must be my DO name", context = "Validation")
 * )
 *
 */
class IsMyName extends Constraint {

  public $message = 'No, "{{ string }}" is not my d.o. name.';

  //public function validatedBy() {
  //  return SomeOtherValidator::class;
  //}

}
```

To start, extend Symfony’s Validator\Constraint class.

Note that in Drupal, constraints are plugins, so your constraint must be in Drupal\my_module\Plugin\Validation\Constraint.

This one is going to check to make sure that a given name is my D.O. handle.

All your Constraint class has to do is define your constraint’s error message. By default, the validator for your constraint is the constraint name + Validator, but you can specify a different validator class in the validatedBy() method.



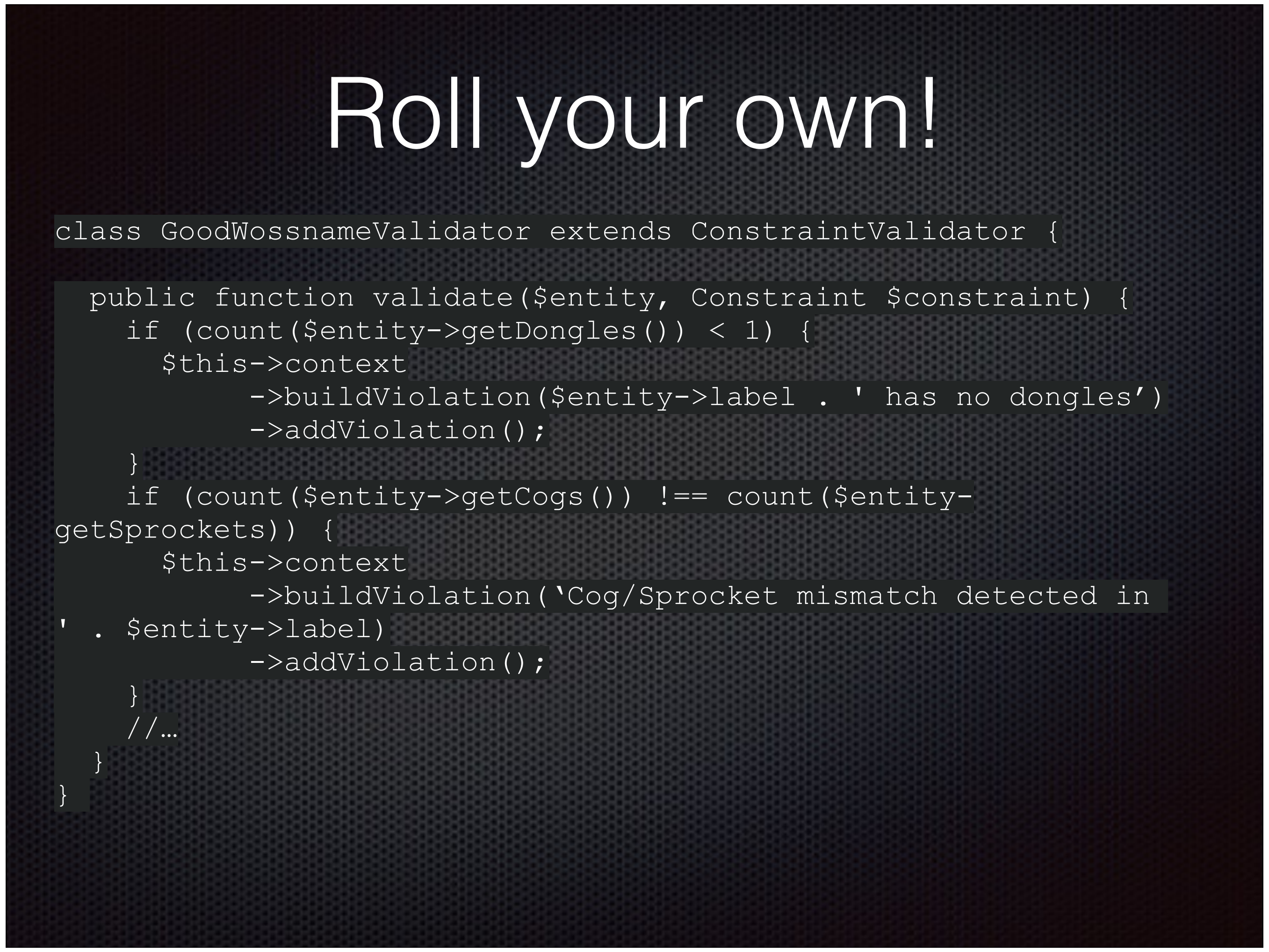
Roll your own!

Create the validator

```
namespace Drupal\surprise_symfony\Plugin\Validation\Constraint;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class IsMyNameValidator extends ConstraintValidator {

  public function validate($value, Constraint $constraint) {
    if (!preg_match('/^sunset_bill$/', $value, $matches)) {
      $this->context->buildViolation($constraint->message)
        ->setParameter('{{ string }}', $value)
        ->addViolation();
    }
  }

}
```

Create the validator class declared that implements validate() to contain your validation logic. I'm going with convention here and using my constraint name + Validator.

The validate() method checks whatever conditions the constraint has to meet and adds a violation for failures.



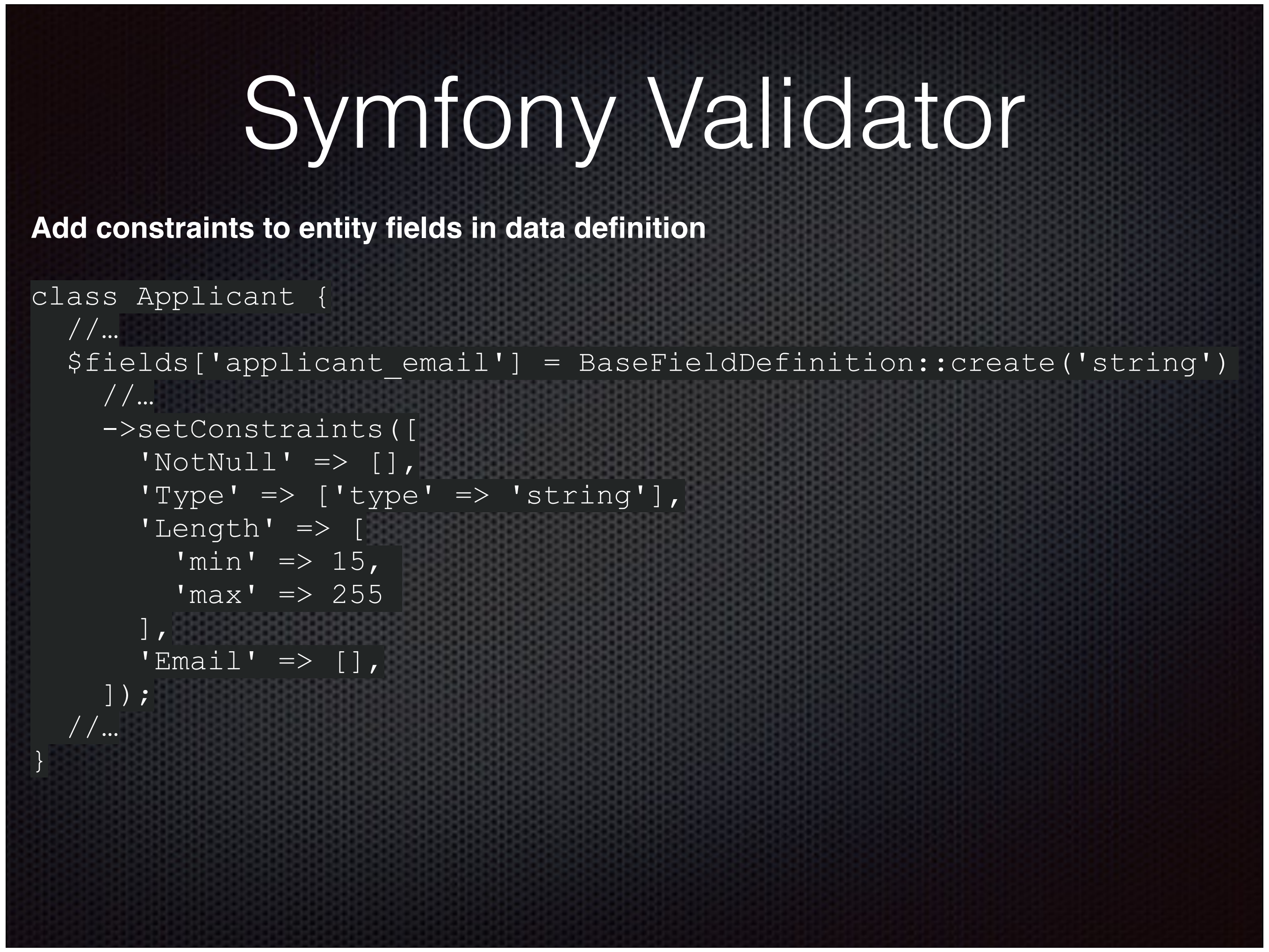
Roll your own!

```
class GoodWossnameValidator extends ConstraintValidator {

  public function validate($entity, Constraint $constraint) {
    if (count($entity->getDongles()) < 1) {
      $this->context
           ->buildViolation($entity->label . ' has no dongles’)
           ->addViolation();
    }
    if (count($entity->getCogs()) !== count($entity-
getSprockets)) {
      $this->context
           ->buildViolation(‘Cog/Sprocket mismatch detected in
' . $entity->label)
           ->addViolation();
    }
    //…
  }
}
```

You can validate an entity by passing an entity to the validate() method.

Say I’ve got an entity called WossName. I want to make sure that every WossName has at least one dongle and an equal number of cogs and sprockets.

Once you have your constraint, you can use it just like one of Symfony’s own.



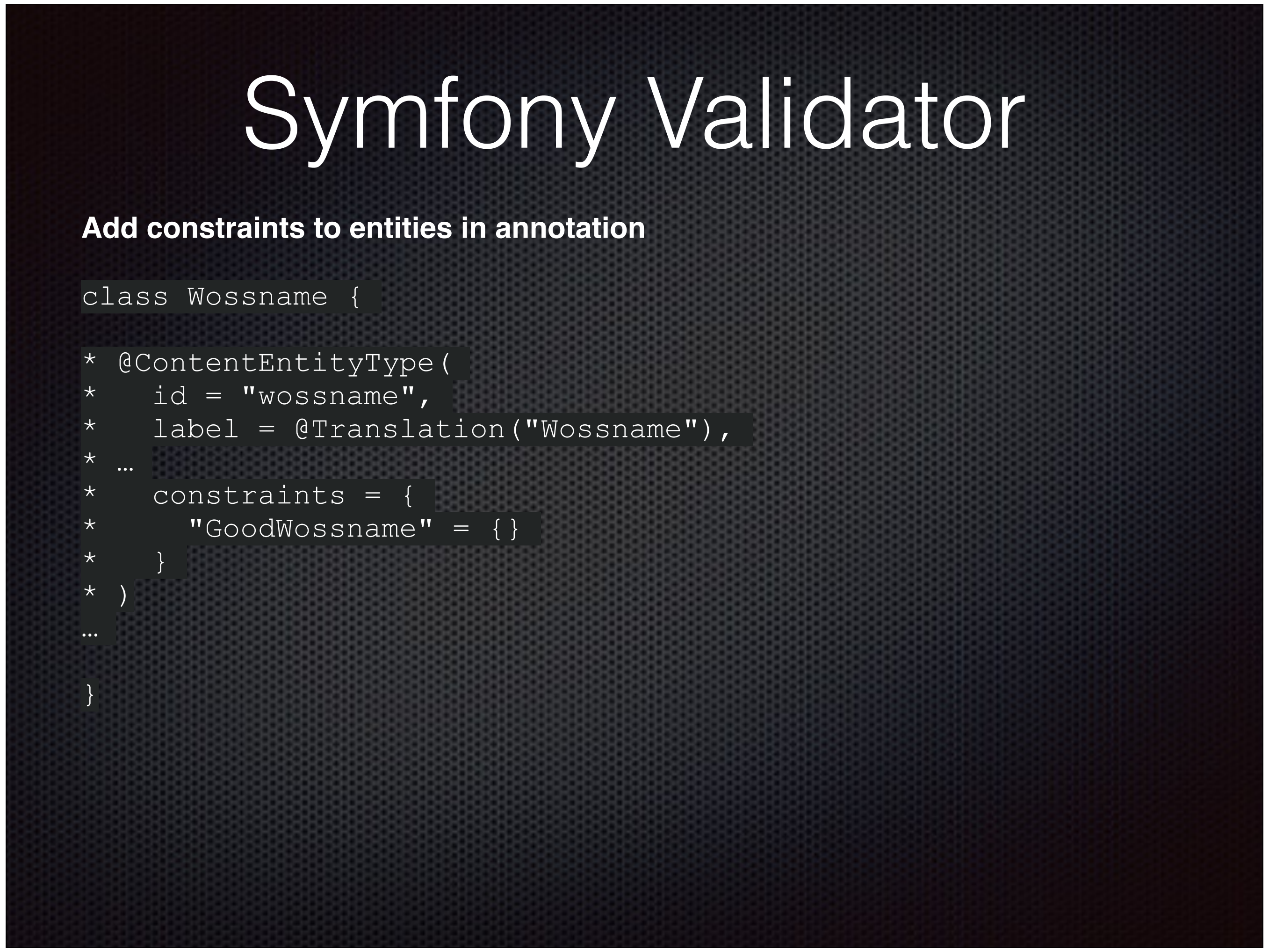
Symfony Validator

Add constraints to entity fields in data definition

```
class Applicant {
  //…
  $fields['applicant_email'] = BaseFieldDefinition::create('string')
    //…
    ->setConstraints([
      'NotNull' => [],
      'Type' => ['type' => 'string'],
      'Length' => [
        'min' => 15,
        'max' => 255
      ],
      'Email' => [],
    ]);
  //…
}
```

In Drupal, you don't even have to pass constraints to a validate() call.

BaseFieldDefinition provides addConstraint() and setConstraints() methods so you can add constraints to any field definition. So, let's check that email string one more time.

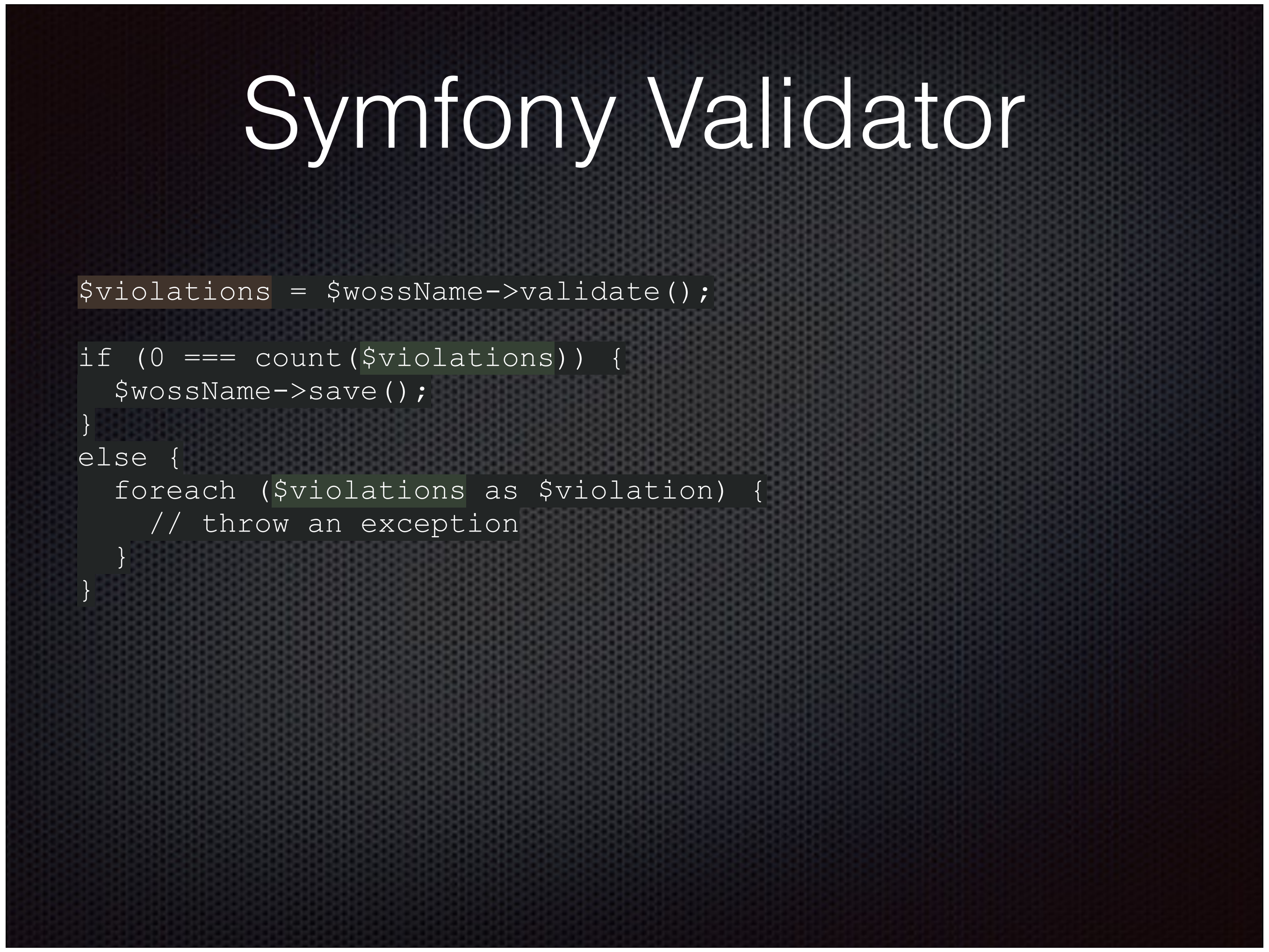


Symfony Validator

Add constraints to entities in annotation

```
class Wossname {

* @ContentEntityType(
*   id = "wossname",
*   label = @Translation("Wossname"),
* …
*   constraints = {
*     "GoodWossname" = {}
*   }
* )
…

}
```

To add constraints to an entity, include a constraints attribute in your entity annotation



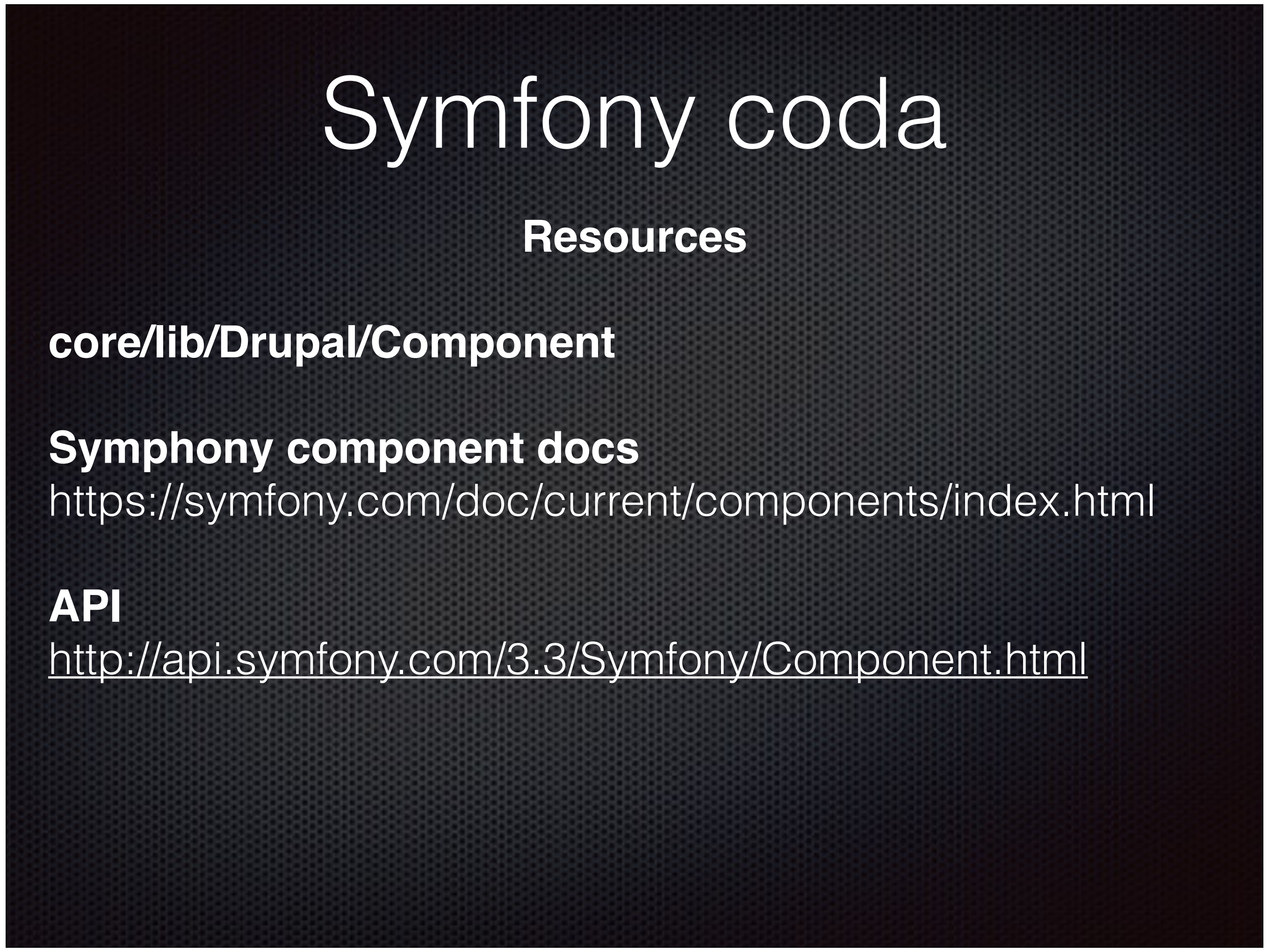
Symfony Validator

```
$violations = $wossName->validate();

if (0 === count($violations)) {
  $wossName->save();
}
else {
  foreach ($violations as $violation) {
    // throw an exception
  }
}
```

This is huge. It means is that, as of D8, validation belongs to entities, not forms. You can tell your entities how to validate themselves rather than relying on some other code to validate them.

And this means you can validate anywhere, any time you're creating an entity, whether it's in a form or in code somewhere. And, indeed, if you look at the validateForm() method in Drupal's ContentEntityForm, all it does is call $entity->validate();

And I think that's a really beautiful thing to end on, 'cause who doesn't need a little validation now and then?


Symfony coda

Resources

core/lib/Drupal/Component

Symphony component docs
https://symfony.com/doc/current/components/index.html

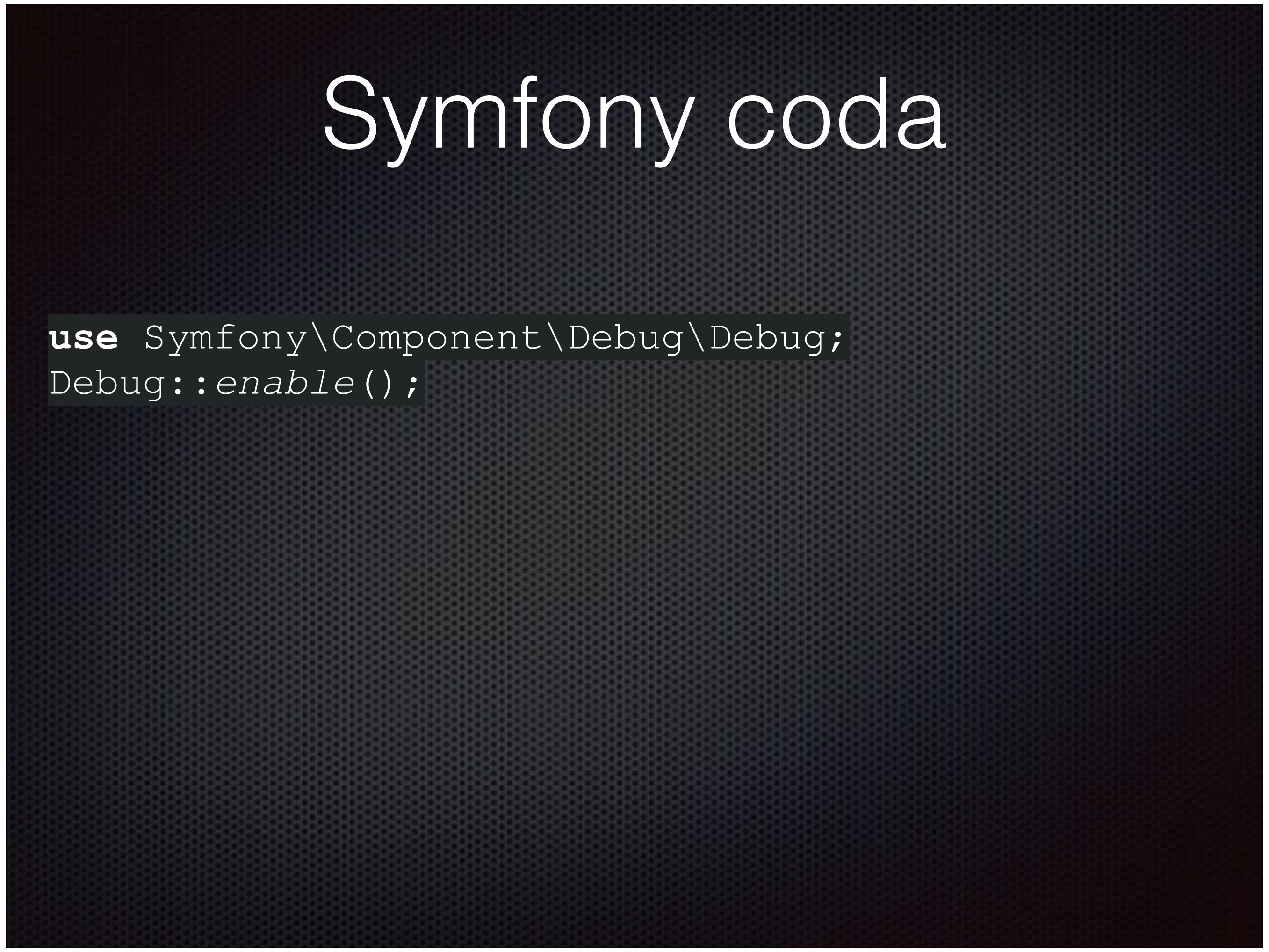
API
http://api.symfony.com/3.3/Symfony/Component.html

Again,this has really just been a bit of surface-scratching. There's enough on just about all of these for an entire presentation. I hope I've at least piqued some curiosity and that some of you will be looking into ways to incorporate all of this rich Symfony goodness into your own code. Maybe we'll even get some deep dives on some of this stuff at DrupalCamp ATL next year.

If you're interested in learning more:

Drupal wraps some of these components and adds a few more, which you can find in your docroot's core/lib/Drupal/Component directory.

I also encourage you to check out the Symfony component and API docs. They really are very good. Lots of detail and easy to read.



Symfony coda

```
use Symfony\Component\Debug\Debug;
Debug::enable();
```

As a parting gift, I'll leave you with one more code snippet. You can add this to your settings.local.php to make your error output more interesting. I've found it even catches some things that Drupal doesn't.

So, what is Drupal hidin'? If it's a little less than it was an hour ago, then

Mischief managed.

Thank you all for being here today.

I think we still have a few minutes for questions or if there are any Symfony gurus out there who want to correct any errors.